Follow the Money: Verifying Smart Contracts with Few False Positives

Anonymous Author(s)

ABSTRACT

In this work, we show how to identify vulnerabilities of Ethereum smart contracts through symbolic execution. The novelty is in reducing false alarms by following the money.

ACM Reference Format:
Anonymous Author(s). 2018. Follow the Money: Verifying Smart Contracts with Few False Positives. In *Proceedings of 40th International Conference on Software Engineering (XXX'2018).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

2 MOTIVATING EXAMPLES

In this section, we present multiple sample vulnerabilities which can be detected using our symbolic execution engine. For each sample, we show the smart contract code, explain how the attack is conducted, particularly, the trace that is executed during the attack and lastly how our engine identifies the attack.

2.1 Sample 1

The first sample contract is named *Rubixi*[1]. The property which is violated by this contract is: *the recipient of a money transaction should not be directly modifiable.*

At some point during the development of *Rubixi*, its name was changed from *DynamicPyramid* to *Rubixi*. However, the authors forgot to change the name of the constructor accordingly. The constructor then became a function named *DynamicPyramid* which is invocable by anyone. Because function *DynamicPyramid* updates the owner's address, which is used as the recipient of a ETH-transferring command in function *collectFeesInEther*, an attacker is allowed to change the recipient arbitrarily and withdraw all money via function *collectFeesInEther*. Shown in Figure 1.

For this vulnerability, the attack is composed of the following steps:

```
1. contract toyRubixi{
2.     uint private collectedFees = 0;
3.     address private creator;

4.     function DynamicPyramid(){creator = msg.sender;}

5.      function collectFeesInEther(uint _amt) onlyowner payable{
6.                _amt *= 1 ether;
7.                if (_amt > collectedFees) _amt = collectedFees;
8.                if (collectedFees == 0) revert();
9.                creator.transfer(_amt);
10.                collectedFees -= _amt;
        }
    }
```

Figure 1: Example toyRubixi contract

(1) First, call function *DynamicPyramid* in line 4 to change the creator to some malicious address.
(2) Then, invoke the *collectFeesInEther* function in line 5, if the amount collected is larger than the contract's *collectedFees*, give all the *collectedFees* to the amount.
(3) Lastly, execute line 9 to transfer the ether to the malicious address.

2.2 Sample 2

The contract shows in Figure 2 (available at[2]) allows a trace which violates the property *there must be an upper bound on the transaction amount.* It is a simplified version of the TheDAO contract. The code is designed such that it is possible to repeatedly execute the ETH-transferring statement so that we can always find a trace sending more money than a given upper bound.

An attack can be conducted using the transaction (written in the form of an Ethereum program named malloryAttack) shown in Figure 3. The attack is composed of the following steps.

(1) At line 18, copy *toyDAO* contract address to *malloryAttack* and deploy the *malloryAttack* contract with his own address.
(2) Next, function *MalloryCollect()* is invoked, which in turn calls function *donate()* in contract *toyDAO* at line 22.
(3) Next, function *withdraw()* is invoked.
(4) Lastly, function *suicide()* is invoked.

The following presents the detailed execution of the attack when function *MalloryCollect()* is invoked.

(1) Execute line 22 to call function *donate()* to increase the attacker's credit.
(2) Execute line 23 to call function *withdraw()*.
(3) Execute line 14 which invokes function *call()*.

[1]https://github.com/gaobo89/Contract_code/blob/master/3TypesExamples/AddressPermute/rubixi.sol

[2]https://github.com/gaobo89/Contract_code/blob/master/3TypesExamples/UpperBound_DAO/toyDao.sol

```
1. contract toyDAO{
2.     address owner;
3.     address public participant;
4.     uint public amout;
5.     mapping (address => uint) credit;

6.     function toyDAO() payable {
7.         donate();
        }
8.     function donate() payable returns (bool){
9.         participant = msg.sender;
10.         amout = msg.value;
11.         credit[msg.sender] += msg.value;
12.         return true;
        }
13.     function withdraw() {
14.         if (!msg.sender.call.value(credit[msg.sender])()) {
15.                revert();
            }
16.         credit[msg.sender] = 0;
        }
    }
```

Figure 2: Example toyDAO contract

```
17. contract MalloryAttack{
    //DAO address is the attacked contract's address, here is
    //0x8a0A8d81192264B0E436b57D30f472A6468780EF for test.

18.     toyDAO public dao = toyDAO(DAO address);
19.     address owner;

20.     function MalloryAttack(){owner = msg.sender;}

        //this is where the attack starts
21.     function MalloryCollect() payable {
22.        dao.donate.value(msg.value)();
23.        dao.withdraw();
           suicide(owner);
        }
    //recursion here when call withdraw without credit decrease.
24.     function() payable {
25.        if(dao.balance > 0){
26.            dao.withdraw();
           }
        }
    }
```

Figure 3: Attacking transaction

(4) Because there is no mapping function, the fallback function at line 24 of *malloryAttack* is invoked.
(5) At line 25, if the balance is positive, function *withdraw* is invoked again and the same steps are repeated.

2.3 Sample 3

The following contract[3] violates the property *there must be some non-zero reward for any work done*. The code is designed such that it is possible to set the reward to be zero and then get the work done for free. The code is shown in Figure 4.

The following presents the potential disputes of the contract.

[3]The code is at https://github.com/gaobo89/Contract_code/blob/master/3TypesExamples/TransactionOrderDependance/ToyMmarketplace.sol

```
1. contract Quiz{
2.     address owner;
3.     uint public reward = 2 ether;
4.     uint public unsolved = 10;

5.     function MarketPlace(){
6.         owner = msg.sender;
    }
7.     function updateReward(uint _reward){
8.         if(msg.sender == owner)
9.             reward = _reward;
    }
10.     function solve(string answer) payable returns (uint){
11.         if(answer is false || unsolved <= 0) revert();
12.                unsolved --;
13.                msg.sender.send(reward);
}   }
```

Figure 4: Example TOD disorder contract

```
1. function enter(address inviter) public {
2.     uint amount = msg.value;
3.     if ((amount < contribution) || (Tree[msg.sender].inviter != 0x0)
           || (Tree[inviter].inviter == 0x0)) {
4.         msg.sender.send(msg.value);
5.         return;
    }
6.     addParticipant(msg.sender, inviter);
7.     address next = inviter;
8.     uint rest = amount;
9.     uint level = 1;
10.     while ( (next != top) && (level < 7) ){
11.         uint toSend = rest/2;
12.         next.send(toSend);
13.         Tree[next].totalPayout += toSend;
14.         rest -= toSend;
15.         next = Tree[next].inviter;
16.         level++;
    }
17.     next.send(rest);
18.     Tree[next].totalPayout += rest;
}
```

Figure 5: Example TOD normal contract

(1) An honest participant calls the function *solve* at line 10 with the correct answer, then send the transaction(we call it transactino1) to the Ethereum mining pool to wait the reward transferred back.
(2) The owner calls *updateReward()* function to change the reward to 0 ether, send the transaction(we call it transaction2) to the Ethereum mining pool.
(3) If transaction2 is mined first, the honest participant will get nothing for reponsing this quiz.

2.4 Sample 4

The following[4] shows a contract which is claimed to be vulnerable using existing approaches, which turns out to be false positive.

Function *enter* in Figure 5 is a snippet from Etheramid, it is claimed by Oyente as a TOD case, but the order of their

[4]The code is at https://github.com/gaobo89/Contract_code/blob/master/3TypesExamples/TransactionOrderDependance/Etheramid.sol

execution does not change the outcome of the contract. The following presents the reason.

(1) The *msg.sender.send* function in line 4 is identified by oyente as an ether flow, it returns Ether to the sender if some condition is not met.
(2) The *next.send* function in line 12 and line 17 are identified as another ether flow, it pays out to the previous participants.
(3) Whether the transaction of item1 succeeds or fails doesn't affect the transaction of item2, in wich the recipients and the amount are certain.