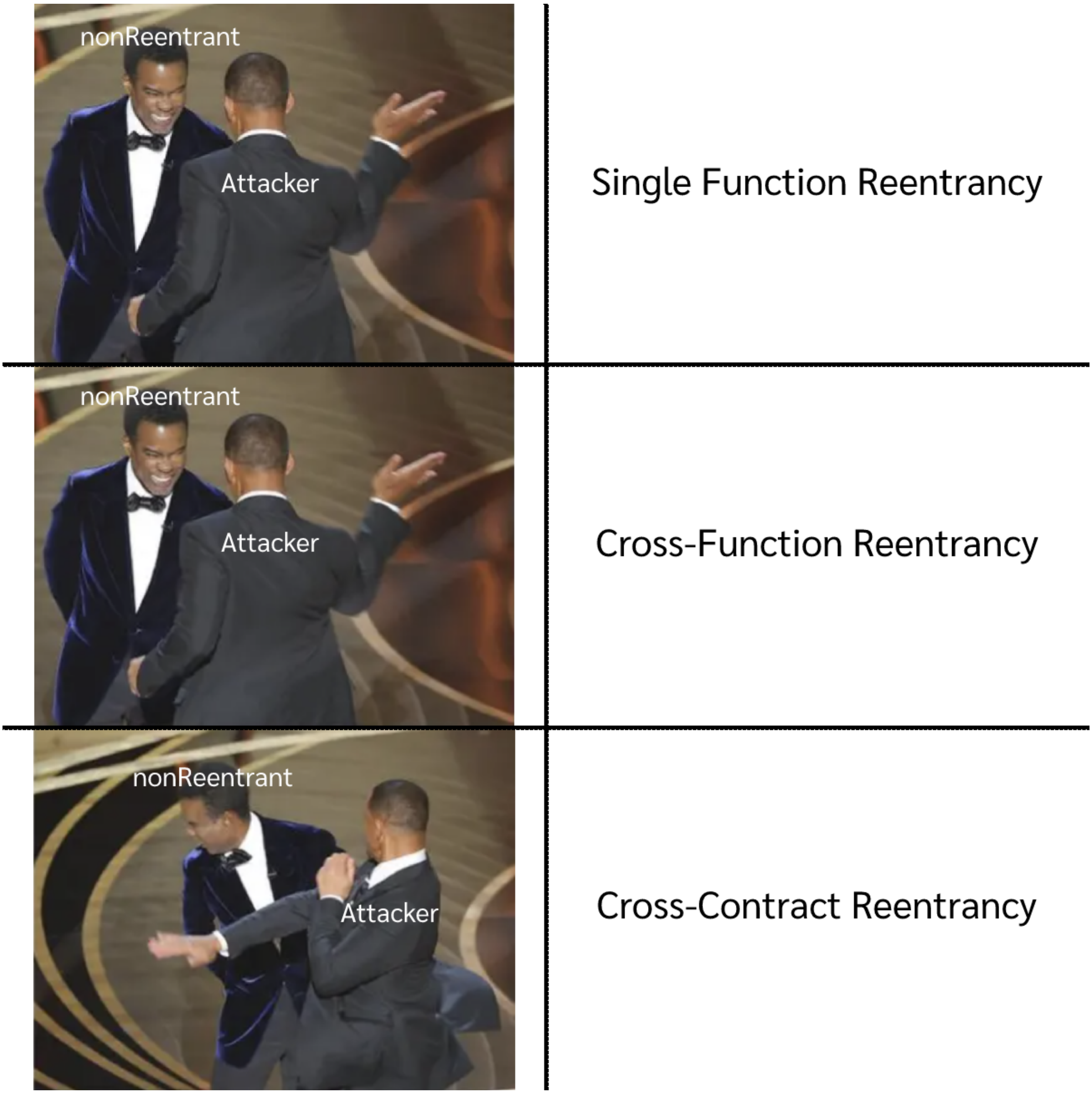
Inspex Follow

Mar 31 · 7 min read · Listen

Save

Cross-Contract Reentrancy Attack


nonReentrant
Attacker
Single Function Reentrancy
nonReentrant
Attacker
Cross-Function Reentrancy
nonReentrant
Attacker
Cross-Contract Reentrancy

Reentrancy attack is one of the most common attacks on EVM-based smart contracts. It

- The DAO attack
- Cream Finance attack

Most reentrancy attacks are done by reentering the same function (Single Function Reentrancy) it is called from; however, there are also other variations that are harder to discover and prevent. In this article, we will show you what **Cross-Contract Reentrancy** is, how impactful can it be, and how can you prevent it. We also have a hands-on lab that you can follow along to learn about this vulnerability more in detail.

Type of Reentrancy

- Single Function Reentrancy
- Cross-Function Reentrancy
- Cross-Contract Reentrancy

The first two variations are commonly found, the examples can be discovered in Consensys's Ethereum Smart Contract Best Practices. What we are going to focus on is the third one, **Cross-Contract Reentrancy**.

Cross-Contract Reentrancy

Cross-contract reentrancy can happen when a state from one contract is used in another contract, but that state is not fully updated before getting called.

The conditions required for the cross-contract reentrancy to be possible are as follows:

1. The execution flow can be controlled by the attacker to manipulate the contract state.
2. The value of the state in the contract is shared or used in another contract.

This kind of vulnerability has been used in multiple past attacks, for example:

- Attack on ValueDefi (7 May 2021)
- Attack on Rari Capital (8 May 2021)

Note: The `router` *smart contract is implemented using UniswapV2's* ***implementation***

Vault.sol

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

interface IRouter {
    function swapExactTokensForTokens(uint amountIn, uint amountOutMin, address[] ca
}

contract Vault is ERC20, ReentrancyGuard {
    using SafeERC20 for ERC20;

    ERC20 public baseToken;
    IRouter public router;

    constructor(ERC20 _baseToken, IRouter _router) ERC20("VaultToken", "VT") public
        baseToken = _baseToken;
        router = _router;
    }

    function shareToAmount(uint256 _share) view public returns (uint256) {
        return _share * baseToken.balanceOf(address(this)) / totalSupply();
    }

    function amountToShare(uint256 _amount) view public returns (uint256) {
        return  _amount * totalSupply() / baseToken.balanceOf(address(this));
    }

    // Deposit tokens into the vault and get shares ($VT)
    function deposit(uint256 _amount) external nonReentrant {
        uint256 total = baseToken.balanceOf(address(this));
        uint256 share = total == 0 ? _amount : amountToShare(_amount);
        _mint(msg.sender, share);
        baseToken.safeTransferFrom(msg.sender, address(this), _amount);
    }

    // Deposit any token and swap it to the base token for depositing to vault
    function swapAndDeposit(uint256 _amount, ERC20 _srcToken, uint256 amountOutMin)
        uint256 beforeTransfer = baseToken.balanceOf(address(this));
        _srcToken.safeTransferFrom(msg.sender, address(this), _amount);
        address[] memory path = new address[](2);
        path[0] = address(_srcToken);
        path[1] = address(baseToken);
        // Approve token for swapping
        uint256 share =  baseTokenAmount * totalSupply() / beforeTransfer;
        _mint(msg.sender, share);
    }

    // Withdraw tokens from the vault by burning shares ($VT)
    function withdraw(uint256 _share) external nonReentrant {
        uint256 amount = shareToAmount(_share);
        _burn(msg.sender, _share);
        baseToken.safeTransfer(msg.sender, amount);
    }

    // Withdraw tokens from the vault by burning shares ($VT) and swap to any token
    function withdrawAndSwap(uint256 _share, ERC20 _destToken, uint256 amountOutMin)
        uint256 amount = shareToAmount(_share);
        address[] memory path = new address[](2);
        path[0] = address(baseToken);
        path[1] = address(_destToken);
        // Approve token for swapping
        baseToken.approve(address(router), amount);
        router.swapExactTokensForTokens(amount, amountOutMin, path, address(msg.send
        // Reset token approval
        baseToken.approve(address(router), 0);
        _burn(msg.sender, _share);
    }

    // Send money somewhere to gain profit
    function work() external nonReentrant {}

```

The users can deposit the base token and get `VaultToken` ($VT) that act as the users' shares of the tokens in the vault.

Without the `work()` and `harvest()` functions implemented, the ratio between the number of shares and the amount of base token in the vault will always be 1:1 unless the base token is manually transferred into the vault.

In the `Vault` contract, the contract itself is safe from reentrancy attack, as a mutex lock (`nonReentrant` modifier) is used, so no attacker can do anything to drain the tokens the value of $VT and the price of token specified in the smart contract.

ICOGov.sol

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.13;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "./GOVToken.sol";
import "./Vault.sol";

contract ICOGov is ReentrancyGuard {

    GOVToken public newToken;
    Vault public vault;
    address treasury;
    uint256 public tokenPrice;

    constructor(GOVToken _newToken, Vault _vault, address _treasury, uint256 _tokenP
        newToken = _newToken;
        vault = _vault;
        treasury = _treasury;
        tokenPrice = _tokenPricePerToken; // price / token
    }

    function buyToken(uint256 _vaultTokenAmount) external nonReentrant {
        // Get token value from share
        uint256 value = vault.shareToAmount(_vaultTokenAmount);
        // Get number of token from value
        uint256 tokenAmount = value / tokenPrice;
        vault.transferFrom(msg.sender, treasury, _vaultTokenAmount);
        newToken.mint(msg.sender, tokenAmount);
    }
}
```

Can you see the flaw in these smart contracts?

Hint: Is it possible to manipulate the share value of $VT?

You can stop here and read through the smart contracts above carefully, or scroll down

Hijacking the Execution Flow

One of the first things that we should notice in the `Vault` contract is that we can control the `_srcToken` address and `_destToken` address parameters in the `swapAndDeposit()` and `withdrawAndSwap()` functions respectively.

Vault.sol

```
pragma solidity 0.8.13; // For syntax highlighting

    // Deposit any token and swap it to the base token for depositing to vault
    function swapAndDeposit(uint256 _amount, ERC20 _srcToken, uint256 amountOutMin)
        uint256 beforeTransfer = baseToken.balanceOf(address(this));
        _srcToken.safeTransferFrom(msg.sender, address(this), _amount);
        address[] memory path = new address[](2);
        path[0] = address(_srcToken);
        path[1] = address(baseToken);
        // Approve token for swapping
        _srcToken.approve(address(router), _amount);
        router.swapExactTokensForTokens(_amount, amountOutMin, path, address(this),
        // Reset token approval
        _srcToken.approve(address(router), 0);
        uint256 baseTokenAmount = baseToken.balanceOf(address(this)) - beforeTransfe
        uint256 share =  baseTokenAmount * totalSupply() / beforeTransfer;
        _mint(msg.sender, share);
    }

        // Withdraw tokens from the vault by burning shares ($VT) and swap to any to
    function withdrawAndSwap(uint256 _share, ERC20 _destToken, uint256 amountOutMin)
        uint256 amount = shareToAmount(_share);
        address[] memory path = new address[](2);
        path[0] = address(baseToken);
        path[1] = address(_destToken);
        // Approve token for swapping
        baseToken.approve(address(router), amount);
        router.swapExactTokensForTokens(amount, amountOutMin, path, address(msg.send
        // Reset token approval
        baseToken.approve(address(router), 0);
        _burn(msg.sender, _share);
    }
```

or burned after the swapping is done.

Moreover, in the `swapAndDeposit()` function, the `_srcToken.approve()` function is called. As we can control the address of `_srcToken` , it is possible to hijack the execution flow before and after the token is swapped!

Using States from Another Contract

From the `ICOGov` contract, the `shareToAmount()` function from the `Vault` contract is called in the `buyToken()` function to determine the value of $VT in terms of the base token.

ICOGov.sol

```
pragma solidity 0.8.13; // For syntax highlighting

    function buyToken(uint256 _vaultTokenAmount) external nonReentrant {
        // Get token value from share
        uint256 value = vault.shareToAmount(_vaultTokenAmount);
        // Get number of token from value
        uint256 tokenAmount = value / tokenPrice;
        vault.transferFrom(msg.sender, treasury, _vaultTokenAmount);
        newToken.mint(msg.sender, tokenAmount);
    }
```

ICOGov-Buy.sol hosted with ❤ by **GitHub** **view raw**

The `shareToAmount()` function calculates the token amount using the balance of the base token in the contract and the total supply.

Vault.sol

```
pragma solidity 0.8.13; // For syntax highlighting

    function shareToAmount(uint256 _share) view public returns (uint256) {
        return _share * baseToken.balanceOf(address(this)) / totalSupply();
    }
```

Vault-Share.sol hosted with ❤ by **GitHub** **view raw**

Directly transferring the base token into the contract is one of the options; however, by transferring directly, the token will be shared among the $VT holder, so a part of the funds will be lost.

Combining the Two

Remember what we found out in the previous part? We can hijack the execution flow before and after the swapping of the token!

What if we use the `swapAndDeposit()` function and take control of the execution flow after the base token is successfully swapped and sent into the `Vault` , but $VT is not yet minted? The value of $VT will be inflated at that moment, and we can call `ICOGov.buyToken()` to mint more token using the inflated value!

Let’s take a look at this code again.

Vault.sol

At line 48 (12 in the gist), the token is swapped and transferred back to the vault, this means that after this line, the base token balance will increase, while the total supply is not updated, as $VT is not minted yet.

And at line 50 (14 in the gist), the `approve()` function of `_srcToken` is called. As we can control the address of the token, we can implement a malicious token with a special `approve()` function that performs a reentrant calling to `ICOGov.buyToken()` when called at this specific point.

Let’s Try Hacking It!

As we want to hijack the execution flow using a malicious token contract, let’s

Starting as a standard `ERC20` token, we can create a contract and inherit OpenZeppelin’s `ERC20` contract implementation.

EvilERC20.sol

We want to implement a custom `approve()` function that triggers at a specific point in the `Vault` contract, so we can start with the baseline, the original implementation of the `approve()` function.

We want it to trigger here at line 50 (14 in the gist), and we know that the `Vault` contract will be the caller, and the approval amount will be zero; therefore, these can be used as the condition to trigger the reentrancy to the `ICOGov` contract.

Vault.sol

We can implement the conditional trigger by checking that the address of the `msg.sender` matches the vault address, and the approval amount is equal to zero.

EvilERC20.sol

With the contract above, we can now hijack the execution flow at the location we want to. Next, we need to prepare the $VT needed to buy the token, and call the `ICOGov.buyToken()` function. We can do that by allowing the contract to transfer $VT from attacker's wallet to `EvilERC20` , approve the transfer, then buy the token.

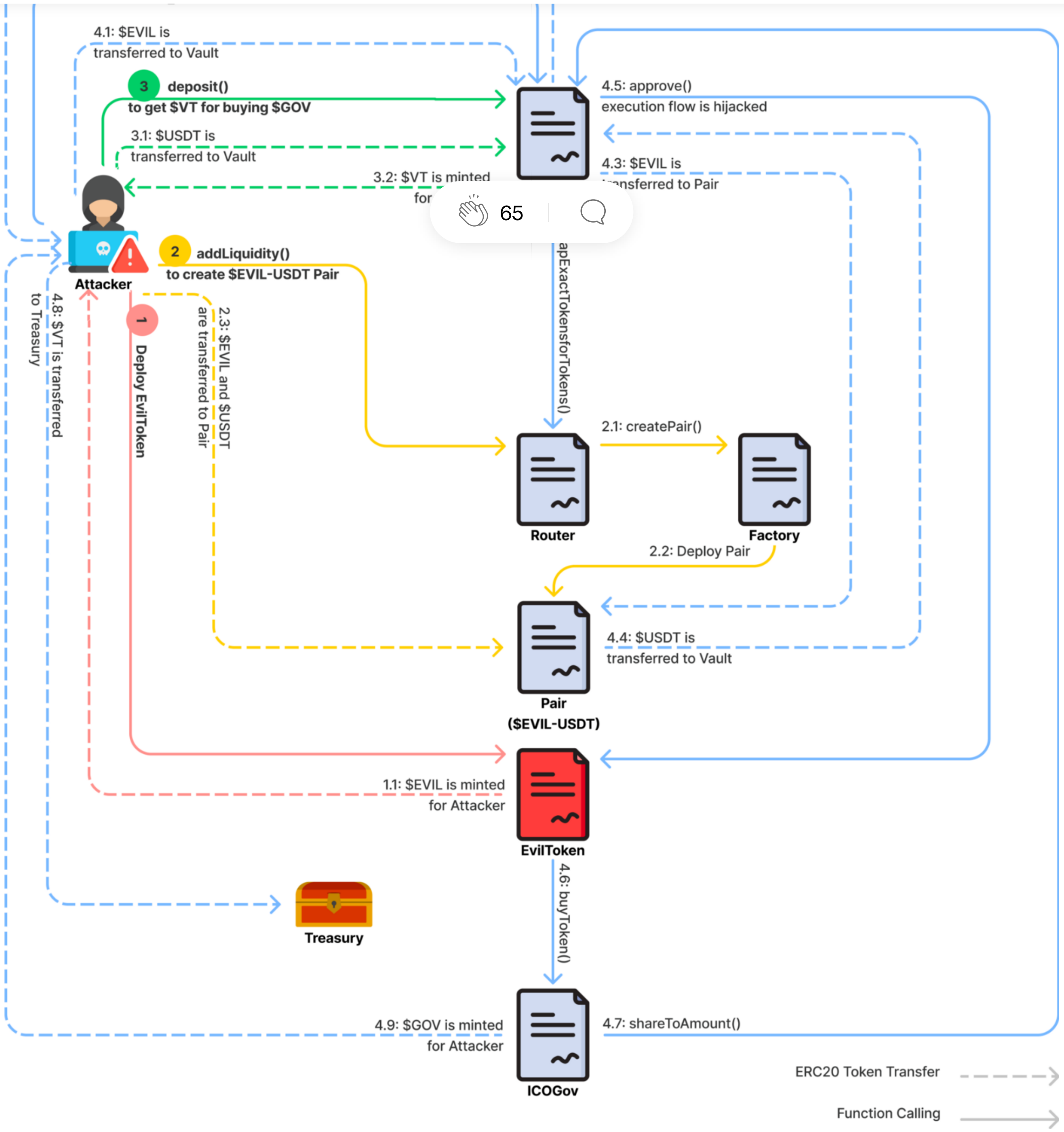
EvilERC20.sol

Don’t forget to transfer the tokens bought back to the attacker’s wallet! And as we also need to prepare the liquidity pool, let’s mint some token to the attacker’s wallet too!

EvilERC20.sol

We are now done with the evil token contract. We can deploy it and perform the attack.

Assuming that the `baseToken` is $USDT, the steps to perform are as shown in the following diagram (some minor details are omitted):


4.1: $EVIL is transferred to Vault
3 deposit() to get $VT for buying $GOV
3.1: $USDT is transferred to Vault
3.2: $VT is minted for
4.5: approve() execution flow is hijacked
4.3: $EVIL is transferred to Pair
Attacker
2 addLiquidity() to create $EVIL-USDT Pair
apExactTokensforTokens()
4.8: $VT is transferred to Treasury
1 Deploy EvilToken
2.3: $EVIL and $USDT are transferred to Pair
2.1: createPair()
Router
Factory
2.2: Deploy Pair
4.4: $USDT is transferred to Vault
Pair ($EVIL-USDT)
1.1: $EVIL is minted for Attacker
EvilToken
4.6: buyToken()
Treasury
4.9: $GOV is minted for Attacker
4.7: shareToAmount()
ICOGov
ERC20 Token Transfer
Function Calling

Cross-Contract Reentrancy Attack diagram

1. Deploy the `EvilToken` contract

2. Prepare $USDT and call `Router.addLiquidity()` to create a $EVIL-USDT Pair (Approve the transfer of $EVIL and $USDT for `Router` first)

3. Call `Vault.deposit()` to get $VT for buying $GOV (Approve the transfer of $USDT for `Vault` first)

With these steps, the value of $VT will be inflated depending on the amount deposited in ste 4, allowing the attacker to buy $GOV at a lower cost.

Furthermore, using a similar method, a flashloan can also be used to largely increase the impact of this vulnerability, for example:

1. Flashloan a large sum of $USDT
2. Deposit $USDT
3. Inflate the value of $VT and perform a reentrant calling to buy $GOV
4. Sell $GOV in the open market
5. Withdraw $USDT used in the inflation of $VT value
6. Return the flashloaned $USDT and profit from the $GOV sold

Solutions to Prevent Reentrancy

For the first two variations, Single Function Reentrancy and Cross-Function Reentrancy, a mutex lock can be implemented in the contract to prevent the functions in the same contrac from being called repeatedly, thus, preventing reentrancy. A widely used method to implem the lock is inheriting OpenZeppelin’s ReentrancyGuard and use the `nonReentrant` modifier.

The better solution is to check and try updating all states before calling for external contract or the so-called “Checks-Effects-Interactions” pattern. This way, even when a reentrant calli is initiated, no impact can be made since all states have finished updating.

Another alternative choice is to prevent the attacker from taking over the control flow of the contract. A set of whitelisted addresses can prevent the attacker from injecting unknown malicious contracts into the contract in this lab.

Nevertheless, the contracts that integrate with other contracts, especially when the states ar shared, should be checked in detail to make sure that the states used are correct and cannot manipulated.

About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

For any business inquiries, please contact us via Twitter, Telegram, contact@inspex.co