SharkTeam

A Vulnerability Perspective Analysis of Move Language Security——logic verification Vulnerability

Nov 18, 2022

SharkTeam, a leading blockchain security service team, offers smart contract audit services for developers. To satisfy the demands of different clients, thesmart contract audit services provide both manual auditing and automated auditing.

We implement almost 200 auditing contents that cover four aspects: high-level language layer, virtual machine layer, blockchain layer, and business logiclayer, ensuring that smart contracts are completely guaranteed and Safe.

In the previous series of "Top 10 Smart Contract Security Threats", SharkTeam summarized and analyzed the top 10 most harmful vulnerabilities in the field of smart contracts based on historical smart contract security incidents.

These vulnerabilities usually appeared in Solidity smart contracts before, so will the same harm exist for the emerging Move smart contracts?

SharkTeam [A Vulnerability Perspective Analysis of Move Language Security] series of courses will discuss and deepen with you. The second lesson [Logic verification Vulnerability].

1 Logical verification vulnerability

The business-related logic design of smart contract development is complex, involving many economic calculations and parameters, and the composability between different projects and protocols is extremely rich, difficult to predict, and very prone to security vulnerabilities.

In Solidity smart contracts, we summarize four types of logic verification vulnerabilities:

(1) The return value is not verified

(2) Unverified related calculation data formulas

(3) Unverified function parameters

(4) Unregulated use of require verification

Similarly, we will analyze whether there are these logic verification loopholes in the Move contract from these four aspects, as well as their possibility and harm.

1.1 Return value not verified

If the return value of the message call is not checked, even if the called function returns an abnormal value, the execution logic will still continue, but the function call does not implement the correct logic, which will cause the entire transaction to not get the correct result, or even It will threaten the security of digital assets.

For example, the call function in the Solidity contract, the functionCallWithValue function is as follows:

```
function functionCallWithValue(
    address target,
    bytes memory data,
    uint256 value,
    string memory errorMessage
) internal returns (bytes memory) {
    require(address(this).balance >= value, "Address: insufficient balance for call");
    (bool success, bytes memory returndata) = target.call{value: value}(data);
    return verifyCallResultFromTarget(target, success, returndata, errorMessage);
}
```

The call function is called in the code. If there is an accident in the execution of the call function, such as a transfer failure, the return value success is false. If the return value is not verified, even if success is false, the transaction will still be executed normally. It's just that this transfer in the transaction didn't work out. Here, the success is verified by require, and if it is false, the transaction will be rolled back (revert).

The call function is a key function of Solidity's dynamic function call, and it is a typical representative of the Solidity language level that is prone to loopholes due to return values. In addition to the call function, at the business level, Solidity contracts often use return values to determine whether the function is

executed successfully, such as the functions in the ERC20 contract:

```
interface IERC20 {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external view returns (string memory);
    function symbol() external view returns (string memory);
    function decimals() external view returns (uint8);
    function totalSupply() external view returns (uint);
    function balanceOf(address owner) external view returns (uint);
    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);
    function transfer(address to, uint value) external returns (bool);
    function transferFrom(address from, address to, uint value) external returns (bool);
}
```

For such functions, it is generally necessary to verify the return value during practical application, otherwise loopholes will be generated, and even the security of digital assets will be threatened.

In addition, according to the actual business logic, the function will return some data required by the business. These data also need to be verified according to the business to further ensure that no accidents occur in the function call, including but not limited to the type, length, and range of the return value. For example, in the above functionCallWithValue function, the verifyCallResultFromTarget function is mobilized to verify the return value. It not only checks the return value success, but also checks and processes the length of retrundata.

```
function verifyCallResultFromTarget(
    address target,
    bool success,
    bytes memory returndata,
    string memory errorMessage
) internal view returns (bytes memory) {
    if (success) {
        if (returndata.length == 0) {
            // only check isContract if the call was successful and the return data is empty
            // otherwise we already know that it was a contract
            require(isContract(target), "Address: call to non-contract");
        }
        return returndata;
    } else {
        _revert(returndata, errorMessage);
    }
}
```

From the language level in the Move contract, due to its static call

characteristics, there is no situation similar to the call function in Solidity that needs to verify the return value. Even if it is necessary to verify whether the function is executed correctly, it is generally used in the spec module. The specification language is verified in Move Prover, and the transaction will be aborted if the verification fails.

From a business perspective, the spec module in the Move contract can also verify the modification of the global data by the function. In addition, unit test functions can be written in the contract to directly perform unit tests on functions to ensure the correctness of function execution. Therefore, the Boolean variable indicating whether the function execution is successful or not is generally not used as the return value. Therefore, the return value of the Move function is mostly actual business data. Whether it needs to be verified or not needs to be determined according to the actual business needs. Test, such as the liquidity function in DEX:

```
public fun liquidity<X: copy + drop + store,
                     Y: copy + drop + store>(signer: address): u128 {
    let order = TokenSwap::compare_token<X, Y>();
    assert!(order != 0, ERROR_ROUTER_INVALID_TOKEN_PAIR);
    if (order == 1) {
        Account::balance<LiquidityToken<X, Y>>(signer)
    } else {
        Account::balance<LiquidityToken<Y, X>>(signer)
    }
}
```

The order of X and Y is different, and the balance that needs to be accessed is also different, and order!=0 needs to be checked.

In general, the static call feature of the Move language, spec modules, and unit tests have greatly improved the security of functions, which is much better than Solidity. However, it does not rule out that the function will have a loophole because the return value is not verified. Therefore, developers need to be more familiar with business and implementation logic, and they need to be cautious when developing.

1.2 Unverified related calculation data

In the process of contract implementation for related businesses, considering the situation is not comprehensive enough, the corresponding business economics formulas and calculation data are not correctly verified, resulting in poor fault tolerance of the contract for special calculation data. for example:

(1) XCarnival security incident

The incident occurred on June 24, 2022, when the NFT lending protocol XCarnival was hacked, resulting in a loss of approximately $3.8 million.

The root cause is that the orderAllowed function called by the borrowAllowed function of the controller contract is incomplete in the verification of the data structure order. It only verifies that the order exists, the address is correct and has not been liquidated. It does not verify whether the NFT in the order has been withdrawn, even if the order The NFT in has been extracted, and the order verification can still pass.

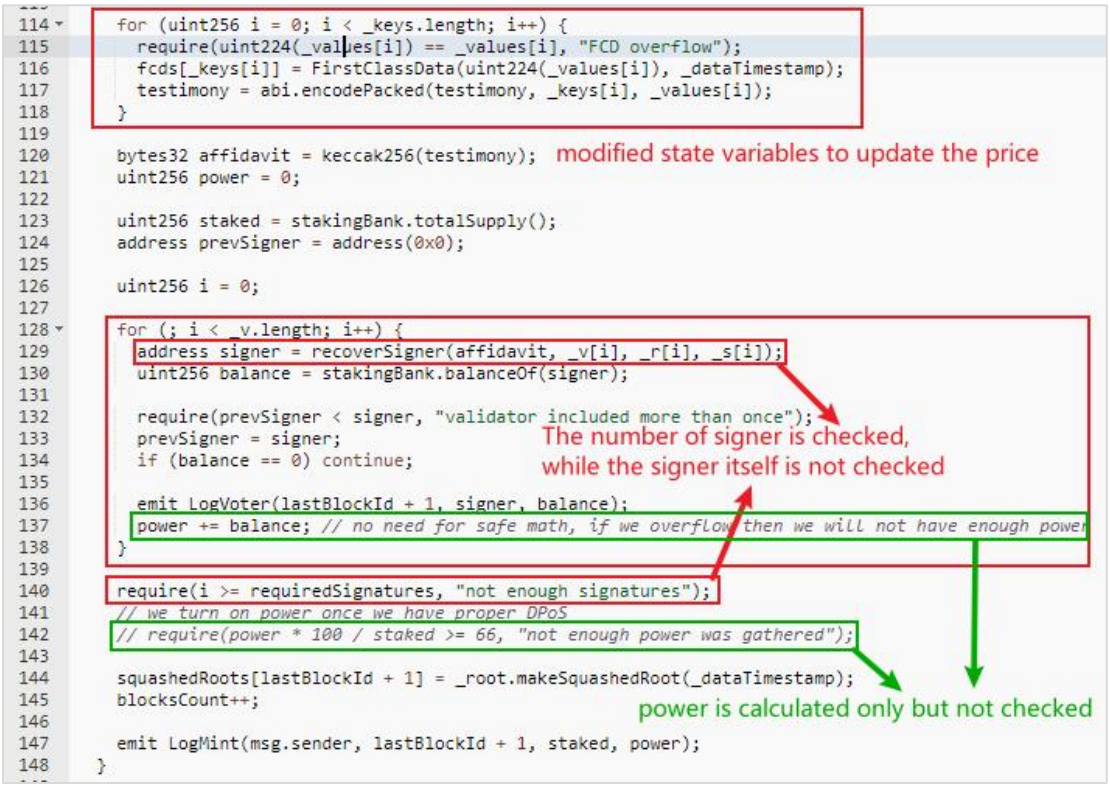
```
59    function borrowAllowed(address xToken, uint256 orderId, address borrower, uint256 borrowAmount) external whenNotPaused(xToken, 3){
60        require(poolStates[xToken].isListed, "token not listed");
61
62        orderAllowed(orderId, borrower);
63
64        (address _collection , , ) = xNFT.getOrderDetail(orderId);
65
66        CollateralState storage _collateralState = collateralStates[_collection];
67        require(_collatera
68        require(_collatera
69
70        address _lastXToke
71        require(_lastXToke
72
73        (uint256 _price, b
74        require(_price > 0
75
76        // Borrow cap of 0
77        if (poolStates[xTo
78            require(IXToken(xToken).totalBorrows().add(borrowAmount) < poolStates[xToken].borrowCap, "pool borrow cap reached");
79        }
80
81        uint256 _maxBorrow = mulScalarTruncate(_price, _collateralState.collateralFactor);
82        uint256 _mayBorrowed = borrowAmount;
83        if (_lastXToken != address(0)){
84            _mayBorrowed = IXToken(_lastXToken).borrowBalanceStored(orderId).add(borrowAmount);
85        }
86        require(_mayBorrowed <= _maxBorrow, "borrow amount exceed");
87
88        if (_lastXToken == address(0)){
89            orderDebtStates[orderId] = xToken;
90        }
91    }

48      function orderAllowed(uint256 orderId, address borrower) internal view returns(address){
49          (address _collection , , address _pledger) = xNFT.getOrderDetail(orderId);
50
51          require((_collection != address(0) && _pledger != address(0)), "order not exist");
52          require(_pledger == borrower, "borrower don't hold the order");
53
54          bool isLiquidated = xNFT.isOrderLiquidated(orderId);
55          require(!isLiquidated, "order has been liquidated");
56          return _collection;
57      }
```

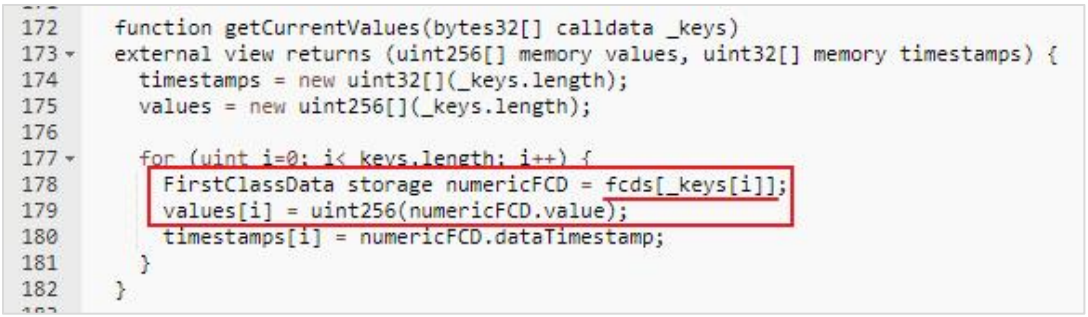
(2) Fortress Loans security incident

The incident occurred on May 9, 2022. Fortress Loans was hacked and lost 1048.1 ETH and 400,000 DAI.

The root cause is that although the submit function verifies the number of signers, it does not verify the signer itself and the calculated data power.



This allows the attacker to call the submit function to modify the state variable fcds, and finally modify the price in the price oracle.



In the end, the attacker used this vulnerability to steal 1048.1 ETH and 400,000 DAI.

There are many similar security incidents, all of which are caused by the lack of data structure for the economic model or the lack of verification of the calculated data inside the function. This type of vulnerability is caused by the fact that the project design and development did not take into account all the circumstances, and its severity varies, and serious ones may even bring great

economic losses to the project, just like the security incident above.

When the Move contract implements various projects, it is also difficult to guarantee that such problems will not occur, especially for new projects. It is hoped that these security incidents that occurred in the Solidity smart contract can give Move developers some warnings, and try to avoid security holes as much as possible during the development process.

1.3 Unverified function parameters

When a function receives parameters it does not automatically verify that the input data attributes are safe and correct. Therefore, when the function is implemented, the parameters need to be verified according to the business needs. If the verification is missing and the latter verification does not meet the business needs, it will cause loopholes and even threaten the security of digital assets.

Take the Superfluid.Finance security incident as an example. The incident occurred on February 8, 2022. The DeFi protocol Superfluid on Ethereum was hacked and lost more than 13 million US dollars.

The root cause is that there are serious logic loopholes in the Superfluid contract. The callAgreement function lacks verification of parameters, which allows the attacker to replace the ctx data constructed by the contract with custom ctx data, which provides an opportunity for the attacker to launch an attack.

```
    function _callAgreement(
        address msgSender,
        ISuperAgreement agreementClass,
        bytes memory callData,
        bytes memory userData
    )
        internal
        cleanCtx
        isAgreement(agreementClass)
        returns(bytes memory returnedData)
    {
        // beaware of the endiness
        bytes4 agreementSelector = CallUtils.parseSelector(callData);

        //Build context data
        bytes memory  ctx = _updateContext(Context({
            appLevel: isApp(ISuperApp(msgSender)) ? 1 : 0,
            callType: ContextDefinitions.CALL_INFO_CALL_TYPE_AGREEMENT,
            /* solhint-disable-next-line not-rely-on-time */
            timestamp: block.timestamp,
            msgSender: msgSender,
            agreementSelector: agreementSelector,
            userData: userData,
            appAllowanceGranted: 0,
            appAllowanceWanted: 0,
            appAllowanceUsed: 0,
            appAddress: address(0),
            appAllowanceToken: ISuperfluidToken(address(0))
        }));
        bool success;
        (success, returnedData) = _callExternalWithReplacedCtx(address(agreementClass), callData, ctx);
        if (!success) {
            revert(CallUtils.getRevertMsg(returnedData));
        }
        // clear the stamp
        _ctxStamp = 0;
    }

    function callAgreement(
        ISuperAgreement agreementClass,
        bytes memory callData,
        bytes memory userData
    )
        external override
        returns(bytes memory returnedData)
    {
        return _callAgreement(msg.sender, agreementClass, callData, userData);
    }
```

In the development of Move contract, it is more necessary to verify the parameters. In Move, the parameters of the function are not only the data required by the business, but also the data required by the authority, such as signer. Move does not have a global variable like msg.sender in Solidity. The authentication of permissions in Move is realized through parameters. For example the following function:

```
public entry fun mint(
    account: &signer,
    dst_addr: address,
    amount: u64,
) acquires MintCapStore {
    let account_addr = signer::address_of(account);

    assert!(
        exists<MintCapStore>(account_addr),
        error::not_found(ENO_CAPABILITIES),
    );

    let mint_cap = &borrow_global<MintCapStore>(account_addr).mint_cap;
    let coins_minted = coin::mint<AptosCoin>(amount, mint_cap);
    coin::deposit<AptosCoin>(dst_addr, coins_minted);
}
```

The account parameter in this function is the originating account of token casting, which must have the authority to mint coins, that is, MintCapStore, similar to the msg.sender in Solidity must be the owner. If this part of the verification is missing, the token can be minted by any account.

In addition, the types of projects in the Move ecosystem are the same as those in the Solidity ecosystem, but the implementation languages are different. Therefore, there is a high possibility that the business logic loopholes in the Solidity contract still exist in the Move contract. Therefore, Move developers should pay attention to these loopholes that have appeared in Solidity contracts when developing projects.

1.4 Unspecified use of require

The require in Solidity is designed to verify the external input of the function, including the parameters input by the caller, the return value of the function, the state change before and after the function execution, etc. If the use of require cannot be standardized, the contract may have loopholes and even threaten the security of digital assets, such as the XDXSwap security incident.

The incident occurred on July 2, 2021. The DeFi project XDXSwap on the Huobi Ecological Chain (Heco) was attacked by a flash loan and lost about 4 million US dollars.

File 1 of 8 : UniswapV2Pair.sol

```
198
199     // this low-level function should be called from a contract which performs important safety checks
200     function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external lock {
201         require(amount0Out > 0 || amount1Out > 0, 'UniswapV2: INSUFFICIENT_OUTPUT_AMOUNT');
202         (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
203         require(amount0Out < _reserve0 && amount1Out < _reserve1, 'UniswapV2: INSUFFICIENT_LIQUIDITY');
204
205         uint balance0;
206         uint balance1;
207         { // scope for _token{0,1}, avoids stack too deep errors
208         address _token0 = token0;
209         address _token1 = token1;
210         require(to != _token0 && to != _token1, 'UniswapV2: INVALID_TO');
211         if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer tokens
212         if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer tokens
213         if (data.length > 0) IUniswapV2Callee(to).uniswapV2Call(msg.sender, amount0Out, amount1Out, data);
214         balance0 = IERC20Uniswap(_token0).balanceOf(address(this));
215         balance1 = IERC20Uniswap(_token1).balanceOf(address(this));
216         }
217         uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
218         uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
219         require(amount0In > 0 || amount1In > 0, 'UniswapV2: INSUFFICIENT_INPUT_AMOUNT');
220         { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
221         // uint balance0Adjusted = balance0.mul(10000).sub(amount0In.mul(25)); // 3-->2
222         // uint balance1Adjusted = balance1.mul(10000).sub(amount1In.mul(25));// 3-->2
223         // require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(10000**2), 'UniswapV2: K');
224         }
225
226         _update(balance0, balance1, _reserve0, _reserve1);
227         emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
228     }
229
```

The fundamental reason is that the lightning loan function realizes the contract, and there is a serious loophole in which the loan is not repaid, resulting in huge losses. This is a serious loophole introduced when the project party forked the Uniswap contract code and modified it, that is, the lack of a require statement for K value verification. The most fundamental reason is the unfamiliarity of the business, which leads to loopholes in the implementation.

In the Move contract, the assert statement and the spec module perform functions similar to require. Similarly, many Solidity ecological projects, including DEX, lending, farm and other types of projects, will appear in the Move ecosystem in the future. The principle and mechanism of Move and Solidity are different, but the business of the project is the same. In view of the numerous pitfalls of Solidity ecological projects and the endless security incidents, although Move has high security, it is still necessary to be cautious when implementing various projects, and try to avoid the same type of loopholes. I hope that the same pitfall will not be stepped on again.

2 Summary

At present, Move is still in the development stage, and the Move ecology is still a certain distance from maturity. There are few developers and lack of developer experience. Not many developers can really develop Move contracts proficiently, so some loopholes at the business level are more likely to occur. This requires the Move contract to be familiar with Move language features and business during the design and development process, so that business loopholes may be less likely to occur.

In addition, Solidity has implemented a large number of business types, such as decentralized exchanges, decentralized lending, income aggregation, leveraged lending, leveraged mining, flash loans, cross-chain transactions, etc. These typical business scenarios need to be realized one by one in the Move ecosystem, and the implementation plan needs to be redesigned based on the

differences between Move and Solidity. In this process, it is relatively easy to have a loophole, just like Solidity has experienced many attacks and a large amount of asset loss in the early days before it gradually matured. Although Move is a highly secure language, no one can guarantee that there are no loopholes. We hope that we can learn from the development process of Solidity, so that the development of Move ecology can avoid detours, reduce losses, and mature faster and more steadily.

About Us

Our vision is to improve security globally. We believe that by building this security barrier, we can significantly improve lives around the world.SharkTeam composes of members with many years of cyber security experiences and blockchain, team members are based in Suzhou, Beijing, Nanjing and Silicon Valley, proficient in the underlying theories of blockchain and smart contracts, and we provide comprehensive services including threat modeling, smart contract auditing, emergency response, etc. SharkTeam has established strategic and long-term cooperations with key players in many areas of the blockchain ecosystem, such as Huobi Global, OKX, polygon, Polkadot, imToken, ChainIDE, etc

SharkTeam

In Math, We Trust!

https://sharkteam.org

https://t.me/sharkteamorg

https://twitter.com/sharkteamorg