SharkTeam

Move language security analysis and contract audit points Fallback Attacks

Dec 2, 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 “Top 10 Smart Contracts Security Threats” series, SharkTeam summarized and analyzed the top 10 vulnerabilities in the smart contract space based on historical smart contract security incidents. These vulnerabilities were usually found in Solidity smart contracts before, so will they be the same for Move smart contracts?

The SharkTeam [Move Language Security Analysis and Contract Audit Essentials] course series will take you step-by-step into the content, including permission vulnerabilities, re-entry vulnerabilities, logical checksum vulnerabilities, function malicious initialization, fallback attacks, manipulation of the prophecy machine, contract upgrade vulnerabilities, sandwich attacks, replay attacks, and proposal attacks. This chapter covers [fallback attack].

Fallback is a mechanism on the blockchain, whether it is a simple transfer transaction or dealing with some more complex logical contract calls, once the transaction fails, the blockchain will revert to the state before the transaction.

Fallback attack refers to deciding whether a transaction is executed or not based on the operation result, and if the operation result does not meet the attacker’s expectation, the attacker will make the transaction fallback.

I. Fallback Attack in Solidity

In Solidity, the reason for the fallback attack is that Solidity provides functions such as require, assert, revert, etc. for judging conditions, and the attacker can use these functions in the contract to determine whether the result of the execution of the calling function is what he expects to get (to his advantage) if so, the transaction will continue to be executed; otherwise, the fallback, that aborts the execution of the transaction and restore the blockchain state.

From the above reasons, it can be seen that an attacker needs to satisfy 3 conditions to launch a fallback attack.

1. randomness. The attacker initiates a transaction, accesses a function in the

contract, and the execution result should have randomness. If the result of each call is the same or the impact on the attacker is the same, there is no need to launch the attack. Because launching a fallback attack will also only cause more damage (extra Gas cost) and no more gain.

2. profitable. Among all the random results, different results have different uses for the attacker, and there exists a certain result that is the most profitable for the attacker. If all outcomes are unfavorable to the attacker, the attacker launching a fallback attack will only cause greater losses (additional Gas costs), which will necessarily not be what the attacker wants.

3. contract access contract. The attacker needs to customize the logic of the attack contract, call the function in the target contract in the attack contract, and then determine whether to continue execution or fallback based on the results. If the EOA account to mobilize the function in the target contract, the result can not be determined in the execution of the transaction, only after the transaction is completed to access the results, when the transaction has been completed, the blockchain state has been updated, it can not be rolled back.

Satisfying the above three conditions creates the possibility for an attacker to launch a fallback attack. Understanding the root causes and conditions of fallback attacks makes it easier to defend against fallback attacks during contract development. In Solidity smart contracts, the most common means of defense is to restrict the contract access to the contract, that is, the contract function related to randomness using modifiers to restrict the caller address, allowing only the EOA account call, not allowing the contract call. The following are two implementations of the modifier:

```
modifier onlyEOA(){
    require(msg.sender == tx.origin, "msg.sender is not tx.origin");
    _;
}

modifier notContract(){
    require(msg.sender.code.length == 0, "msg.sender is a contract");
    _;
}
```

Among various types of ecological projects, GameFi projects are more vulnerable to fallback attacks because there are more scenarios in GameFi projects where randomness is applied to increase the user experience, for example, CryptoZoan and CryptoZoons are two GameFi projects that have suffered from fallback attacks.

II. CryptoZoan Security Incident

CryptoZoan is a GameFi project on top of the Coin Smartchain (BSC) focusing on NFT gameplay and experience, aiming to create a new financial system that combines blockchain and gaming, allowing users to earn while playing. The game enhances the user experience by incorporating a randomization mechanism that allows users to hatch in-game “eggs” to obtain ZOANs, which are randomly assigned a rarity level upon hatching and are attached to that rarity level for life. There are six rarity levels, with level 1 being the lowest and level 6 being the highest, and the higher the level, the higher the value.

The attacker uses a combination of randomness and fallback to make the attacker’s hatching request get the desired result (a rank of 6), destroying the fairness of the game and allowing the attacker to gain the highest benefit.

The attack process is as follows.

(1) Granting authorization to the attack contract for the egg NFT to be hatched.

(2) Call the attack contract to trigger the evolveEgg(uint256) function.

(3) check the rarity, if the rank does not meet the expected (rank 6) then

perform a rollback; (may consume Gas cost, need to weigh the choice to continue.)

(4) Remove the high rank pet with rank 6.

The attack contract code is as follows:

```
function sendTx(address target1, bytes memory callData1) public authorized returns(bytes memory){
    //solium-disable-next-line security/no-call-value
    (bool success, bytes memory returnData) = target1.call{value: 0}(callData1);
    require(success, _getRevertMsg(returnData));
    return returnData;
}

function onlyR6(uint256 id1, address target1, bytes memory callData1) public {
    sendTx(target1, callData1);
    uint256 rare = nft.getRate(id1);
    require(rare >= 6, "under target rare");
}
```

III. CryptoZoons security events

CryptoZoons are tasked with fighting monsters. Players first select the creatures they want to use in battle and equip them with weapons by purchasing these items at MarketPlace. The player will pay a little BNB gas price to fight against the selected enemy, while the winning side will receive a certain number of tokens.

It is the fallback mechanism in the contract that allows the attacker to determine whether or not he has won by the balance at the end of the battle. In case of victory, the transaction is sent for normal execution; in case of failure, a rollback is performed.

The attack process is as follows.

(1) the pet NFT approved authorization to the attack contract.

(2) call the function in the contract to trigger the battle.

(3) check whether the tokens are increased (whether the battle is won).

(4) If the battle fails, the transaction is rolled back; if the battle is won, the transaction is sent and executed.

The attack contract code is as follows:

```
function sendTx(address target1, bytes memory callData1) public authorized returns(bytes memory){
    //solium-disable-next-line security/no-call-value
    (bool success, bytes memory returnData) = target1.call{value: 0}(callData1);
    require(success, _getRevertMsg(returnData));
    return returnData;
}

// if token increase -> win. else revert
function onlyWin(uint256 id1, address target1, bytes memory callData1) external {
    nft.safeTransferFrom(msg.sender, address(this), id1);
    uint256 balanceBefore = token.balanceOf(address(this));
    sendTx(target1, callData1);
    uint256 balanceAfter = token.balanceOf(address(this));
    require(balanceAfter > balanceBefore, "not win");
    nft.safeTransferFrom(address(this), _msgSender(), id1);
    token.transfer(_msgSender(), balanceAfter);
}
```

IV. Fallback attack in Move

The above analysis shows the three necessary conditions for the fallback attack: randomness, profitability, and contract access to the contract. Among them, randomness and profitability are business-level requirements, while contract access to the contract is a code-level requirement. Compare the Move contract with the ecology, which also has ecologies such as NFT and GameFi, and also uses randomness. In addition, functions in Move contracts are generally callable by contracts (modules and scripting programs). Therefore, the conditions for fallback attacks may also exist in Move contracts.

To address the fallback attack in the Move ecology, we propose the following two ideas.

(1) Do not allow contracts (modules and scripts) to be called, similar to how only EOA accounts are allowed to be called in Solidity. In Move, the entry modifier is intended to allow module functions to be called safely and directly like scripts. This allows module writers to specify which functions can be entry points for starting execution. If the entry modifier is private, the function cannot be called by modules and scripts and can only be the entry point to start execution, i.e. it can only be called by command or SDK, etc., and the transaction execution is complete when the returned result is obtained after the call and cannot be rolled back.

(2) Add time lock. After submitting a transaction, the result cannot be read immediately, but needs to wait for a certain period of time, i.e. adding time locking, such as using a transaction buffer queue, which requires waiting for a certain period of time (e.g., a block of time) after the transaction is submitted before it can be executed. Consider also that the execution results of a transaction are not allowed to be queried immediately after the transaction is executed, but need to wait for time locking. Doing so would ensure that the result cannot be determined in the transaction that calls the function, and would no longer constitute a condition for a fallback attack.

About Us

SharkTeam's vision is to fully secure the Web3 world. The team consists of experienced security professionals and senior researchers from around the world, well versed in the underlying theory of blockchain and smart contracts, providing services including smart contract auditing, on-chain analysis, emergency response, and more. We have established long-term partnerships with key players in various areas of the blockchain ecosystem, such as Polkadot, Moonbeam, polygon, OKC, Huobi Global, imToken, ChainIDE, etc.

Twitter: https://twitter.com/sharkteamorg

Discord: https://discord.gg/jGH9xXCjDZ

Telegram: https://t.me/sharkteamorg

SharkTeam

In Math, We Trust!

https://sharkteam.org

https://t.me/sharkteamorg

https://twitter.com/sharkteamorg