SharkTeam

A Vulnerability Perspective Analysis of Move Language Security—— Reentrancy Attacks and Permission Vulnerabilities

Nov 14, 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 first lesson [Permission Vulnerabilities and Reentrancy Attacks].

1 Permission vulnerability

A permission vulnerability refers to a flaw in the application's authorization check, which allows an attacker to bypass the permission check by using some methods to access or operate other users or higher permissions after obtaining a low-privileged user account.

Permission loopholes in smart contracts are related to critical logic, such as minting tokens, withdrawing funds, changing ownership, etc.

In Solidity contracts, permission vulnerabilities mainly include the following types:

- Function default visibility
- Lack of modifier validation or validation errors or loopholes
- tx.origin authentication
- Initialization function problem
- call self-destruct (selfdestruct)

1.1 Functions are visible by default

In Solidity, there are 4 kinds of permissions for functions, namely: private, internal, external and public.

private: private function, only visible inside the current contract, invisible to transactions, other contracts and derived contracts, only supports internal calls of the current contract;

internal: internal function, only the current contract and derived contracts are visible, transactions and other contracts are not visible, only the internal calls of the current contract and derived contracts are supported;

external: external functions, only other contracts and transactions are visible, the current contract and derived contracts are not visible, only other contracts and external calls that initiate transactions are supported;

public: public functions, visible to all accounts, including transactions, other contracts, current contracts, derived contracts (contracts that inherit the current contract), support other contracts and external calls to initiate transactions, and internal calls of current contracts and derived contracts.

If a function in a contract in Solidity does not declare visibility, it defaults to public, that is, all accounts are visible and have the highest authority. If it is not set properly, the functions that should be set as internal/private will become public functions by default, which will bring great threats to the contract, and even threaten the asset security and key business of the contract.

Compared with Solidity, the Move function has richer and more flexible visibility, including public, entry/script, friend and private.

private: The current Module is visible, and only functions in the current module can be called;

public(friend): Module-level restrictions, only the current module and friend modules are visible, and can only be called by functions in the current module and friend modules;

public: visible to all modules and scripts, can be called by any function defined in any module and script, and cannot be used for transactions;

entry/public(script): Visible to all modules and scripts, can be called by any function defined in any module and script, and can also be used as an entry for transactions.

The function visibility in Move is private by default, which well protects the privacy of functions and is more secure than Solidity. Although the default visibility of the Move function is security, because the visibility of Move is more flexible, it is also necessary to guard against the risk of incorrect visibility declaration.

1.2 Lack of modifier validation or validation errors or loopholes

Solidity functions use the keyword modifier to declare some permission requirements for calling the function. When calling a function, the caller needs to satisfy the permissions specified in modifier to be able to call the function. If the modifier verification conditions are missing or there are errors or loopholes in the verification conditions, this allows the caller to bypass the permission verification to call the function, which will bring certain threats to the contract and assets.

Although the Move contract has no modifier, it also requires specific permission verification. For example, the Move function uses the acquires key to declare that the resource has been acquired. In addition, the verification of some permissions in Move is written in the Move specification language. All of

these may bring certain risks to the Move contract.

1.3 tx.origin authentication

Solidity contracts have the concept of contextual calls, including tx.origin and msg.sender.

- tx.origin represents the original caller, usually the address of EOA;
- msg.sender represents the current caller, usually obtains the address of the upper-level caller, which can be the EOA address or the contract address.

The contract uses the global variable tx.origin as the authentication credential, which is easy for attackers to trick the owner into signing the attack transaction through social engineering, thereby bypassing the owner authentication.

The Move contract has no concept of contextual invocation. The Move contract function is called directly through the address, module name, and function name, and there is no concept of the caller. The calling permissions of functions are also set through function visibility, without the need to carefully verify the identity of the caller. Therefore, there is no such vulnerability in Move.

1.4 Initialization function problem

In Solidity, the initialization function does not limit the caller or the number of times, and it is easy to be called by an attacker to customize the state variables. On the surface, the attacker just took advantage of the vulnerability of the initialization function and modified the state variable. The underlying fact is the modification permission vulnerability of state variables, that is, state variables can be modified, and the function of modifying state variables can be exploited by attackers to steal digital assets in the contract.

From this point of view, Move is safer than Solidity, because the digital assets in Move are unique resource types that cannot be arbitrarily modified or lost,

let alone copied. In addition, the resources in Move can only be modified by the module that created it, and the resources saved in the user account can only be modified by the user.

1.5 Calling selfdestruct

The selfdestruct(address payable addr) function in the Solidity contract can destroy the current contract and send the balance of the current contract to the specified address addr. Therefore, if there is no permission limit when calling selfdestruct, any account can call this function to destroy the contract, causing huge losses of the remaining Tokens in the contract, and further causing contract denial of service attacks, which will cause serious consequences.

In contrast, there is no function with self-destruction function in the Move contract, so there is no such vulnerability in Move.

Although Move does not have a self-destruction function, resources in Move can be destroyed. Each resource in Move represents a digital asset, and every wrong destruction of an asset is a huge loss. The unprovoked destruction of resources will bring security risks to the contract and users. Therefore, it is necessary to strictly control the destruction authority of Move resources and cut off the possibility of arbitrary destruction of resources.

2 Reentrancy Attack

Reentrancy, literally, means that during the function call process, the current function is called again, that is, the execution re-enters the current function. Solidity is a dynamic language, and after calling the call function, an unnamed callback function fallback is executed, which can be a custom function. These features provide conditions for reentrancy. From this perspective, reentrancy is not a vulnerability. Dynamic invocation and even reentrancy make Solidity smart contracts flexibly implement some special services in DeFi, such as flash loans.

Taking the flash loan of Uniswap V2 as an example, its implementation principle is to transfer money first in the swap function, then dynamically call the business function, and finally repay (including handling fees), and check the repayment amount through K value verification, so as to ensure that the balance is satisfied Consistency of economic models and contract states.

```
// this low-level function should be called from a contract which performs important safety checks
function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external lock {
    require(amount0Out > 0 || amount1Out > 0, 'UniswapV2: INSUFFICIENT_OUTPUT_AMOUNT');
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
    require(amount0Out < _reserve0 && amount1Out < _reserve1, 'UniswapV2: INSUFFICIENT_LIQUIDITY');

    uint balance0;
    uint balance1;
    { // scope for _token{0,1}, avoids stack too deep errors
    address _token0 = token0;
    address _token1 = token1;
    require(to != _token0 && to != _token1, 'UniswapV2: INVALID_TO');
    if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer tokens
    if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer tokens
    if (data.length > 0) IUniswapV2Callee(to).uniswapV2Call(msg.sender, amount0Out, amount1Out, data);
    balance0 = IERC20(_token0).balanceOf(address(this));
    balance1 = IERC20(_token1).balanceOf(address(this));
    }
    uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
    uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
    require(amount0In > 0 || amount1In > 0, 'UniswapV2: INSUFFICIENT_INPUT_AMOUNT');
    { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
    uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
    uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
    require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(1000**2), 'UniswapV2: K');
    }

    _update(balance0, balance1, _reserve0, _reserve1);
    emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
}
```

flashloan: transfer tokens

call function including flashloan repayment

check repayment by K vaule

If the flash loan business has a demand for token exchange, and then the business function calls the swap function again, reentrancy occurs during the function call process. If there is no additional profit, it should not be a reentrancy attack, because this is only to meet the actual business needs.

The actual situation is that it is difficult to define whether a reentrancy is an attack, because reentrancy can easily break the atomic operation of "calling external functions and modifying state variables". Therefore, the swap function uses modifier lock to prevent reentrancy.

From this point of view, the root cause of reentrancy attacks is that the atomic operation of "calling external functions and modifying state variables" is

destroyed when reentrancy calls external functions, such as calling external functions twice, but only modifying state variables once.

Therefore, for reentrancy vulnerabilities, we have a suggestion to adopt the "check-validate-interaction" mode, that is, modify the state variable first, and then execute the external function call, which can ensure that the state variable will be modified every time the external function is called.

But in practice, it is possible to do this for simple functions, such as the function that only calls the external function once and modifies the state variable once. If the function is very complex, including multiple external function calls, and the business related to the state variable update check is also very complex, this solution is not so effective, and developers need to use other methods to defend against reentrancy attacks. For example, adding an anti-reentrancy lock does not allow function reentrancy at all.

If from the perspective of the root cause, we can consider all transactions that obtain additional benefits by breaking the atomicity of "calling external functions and modifying state variables" as reentrancy attacks, then function reentrancy is just for reentrancy attacks a possibility offered. Next, let's consider what role dynamic calls play in reentrancy attacks?

What is a dynamic call? How is it different from static calls?

In a Solidity contract, the call of an external function (call) is determined by the address of the calling contract and the function signature (methodId) of the calling function, and is executed in the context of the function during execution, and the contract before execution is compiled. At this stage, the code of the external contract will not be loaded and compiled. At this time, the external function call has only one contract address and function signature, and its calling logic is dynamically unknown. It will only be dynamically determined according to the actual passed parameters during execution. The actual function to be executed, that is, the passed parameters are different, and the

executed function is different.

In contrast, static calling is to load and compile each function during the compilation process, and the execution method and logic of the entire function are statically known.

For a dynamically executed function call, only when it is actually executed can it be truly determined whether its logic destroys the atomicity of "calling external functions and modifying state variables". Until then, it is impossible to check and determine.

This situation may also occur with static calls, but it can be checked before compilation and compilation. The more fundamental reason is that there are loopholes in the contract business logic, which leads to the destruction of "calling external functions and modifying state variables" atomicity problem. This situation is very easy to detect.

In addition, Solidity will execute the callback function fallback() after dynamically calling the external function. The callback function can be customized. This reentrancy attack provides the necessary conditions.

Comparing dynamic calls and static calls, we conclude the following conclusions:

Dynamic invocation is a necessary condition for reentrancy attacks, and reentrancy attacks are possible only when the contract is executed dynamically.

In statically called contracts, reentrancy attacks will not occur, but there may be similar business implementation vulnerabilities, but they are no longer reentrancy vulnerabilities.

Through the above analysis, we reorganize the concept of reentrancy attack, as follows:

Re-entrancy attacks in smart contracts refer to the fact that attackers use the characteristics of dynamic calling of external functions to destroy the atomicity

of "calling external functions and modifying state variables" by customizing callback functions, thereby obtaining additional income attacks.

Move is a statically called language and does not support the callback function fallback(), so reentrancy attacks are impossible in the Move contract, but this does not mean you can sit back and relax. Through the above analysis, although Move does not have the conditions for launching reentrancy attacks, there may be business logic vulnerabilities with similar reasons. Although such vulnerabilities are relatively low-level and easy to be discovered, they may still exist, especially for beginners. and careless developers. The uniqueness and storage characteristics of Move resources can also avoid some asset losses caused by business loopholes to a certain extent.

The Move resource is unique. The Move Token is a type of structure that represents a resource in which the amount of tokens is kept. Quantity is a value type, a specified number of tokens are resources, and the issuance of tokens is based on resource-type tokens as atomic units, which cannot be minted, destroyed, or copied at will. The token transfer of Move is like a simple numerical addition and subtraction in Solidity. The most important thing about Move token is to realize the transfer through the minting and destruction of resources.

Move's token resources are stored in the user's personal account after minting, and only the user has the right to dispose (transfer, etc.) these token resources. In Solidity contracts, the token balance is a value stored in a state variable in the token contract, and these tokens can be copied and transferred if the contract has loopholes.

It can be seen that the Move language ensures the security of user assets from multiple levels.

3 Summary

At present, Move is still in the development stage, and the Move ecosystem is

still far from maturity. There are few developers, and the developers are inexperienced. There are not many people who can really proficiently develop Move contracts, so it is more prone to some loopholes at the business level. This requires that the Move contract must be familiar with the Move language features and business during the design and development process, so as to avoid business loopholes.

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, and cross-chain transactions. These typical business scenarios need to be implemented 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 some vulnerabilities, just like Solidity has experienced many attacks and losses of a large number of assets in the early days before it gradually matured. Although Move is a highly secure language, no one can guarantee that there will be no loopholes. We hope that we can learn from the development process of Solidity, so that the development of the Move ecology will take less detours and less losses, and move towards maturity 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