SLOWMIST

Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2023.04.14, the SlowMist security team received the Nexvault team's security audit application for nexvault wallet contract, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

This is a multisignature wallet. Allows multiple parties to agree on transactions before execution.

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	The DoS issue	Denial of Service Vulnerability	Suggestion	Confirmed
N2	The reminder to use the factory contract	Others	Suggestion	Confirmed

NO	Title	Category	Level	Status
N3	Batch execution risk	Others	Suggestion	Confirmed
N4	Unchecked return value	Others	Suggestion	Confirmed
N5	Risk of front-running attack	Reordering Vulnerability	High	Fixed
N6	Parameter type error	Others	Suggestion	Confirmed

4 Code Overview

4.1 Contracts Description

Codebase:

Audit Version

Project address: https://github.com/nexvault/nexvault-wallet-contract

Commit: 843c209264e71096449f98db3436e90f5dc54bde

The main network address of the contract is as follows:

MultiSigWalletFactory:

https://etherscan.io/address/0x258e64d7044Ed20513656df4Ba1fDfC12A1633a7#code

MultiSigWalletImplementationBeacon:

https://etherscan.io/address/0xC708ef3703Fc6861A785CfCd8F4E227101F5C9c2#code

MultiSigWalletImplementation:

https://etherscan.io/address/0x2cF26caF5F1B457de18C19A08C6597943aA246B2#code

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

MultiSigWalletImplementation			
Function Name	Visibility	Mutability	Modifiers
<Receive Ether>	External	Payable	-
<Fallback>	External	Payable	-
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	validRequirement
addOwner	Public	Can Modify State	onlyWallet ownerDoesNotExist notNull validRequirement
removeOwner	Public	Can Modify State	onlyWallet ownerExists
replaceOwner	Public	Can Modify State	onlyWallet ownerExists ownerDoesNotExist
changeRequirement	Public	Can Modify State	onlyWallet validRequirement
submitTransaction	Public	Can Modify State	-
confirmTransaction	Public	Can Modify State	ownerExists transactionExists notConfirmed
_confirmTransaction	Internal	Can Modify State	ownerExists transactionExists notConfirmed
revokeConfirmation	Public	Can Modify State	ownerExists confirmed notExecuted
executeTransaction	Public	Can Modify State	ownerExists confirmed notExecuted
_executeTransaction	Internal	Can Modify State	notExecuted
external_call	Internal	Can Modify State	-
isConfirmed	Public	-	-
addTransaction	Internal	Can Modify State	notNull
_addTransaction	Internal	Can Modify State	notNull

MultiSigWalletImplementation			
getConfirmationCount	Public	-	-
getTransactionCount	Public	-	-
getOwners	Public	-	-
getConfirmations	Public	-	-
getTransactionIds	Public	-	-
hashTransaction	Public	-	-
getTransactionMessage	Public	-	-
isVerify	Public	-	-
batchSignature	Public	Can Modify State	-
multiCall	Public	Can Modify State	onlyWallet

MultiSigWalletFactory			
Function Name	Visibility	Mutability	Modifiers
createMultiSigWallet	Public	Payable	-
calculateMultiSigWalletAddress	Public	-	-
createMultiSigWalletWithTransaction	Public	Payable	-

MultiSigWalletProxy			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
<Fallback>	External	Payable	-
<Receive Ether>	External	Payable	-
_delegate	Internal	Can Modify State	-

MultiSigWalletProxy			

MultiSigWalletImplementationBeacon			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-

4.3 Vulnerability Summary

[N1] [Suggestion] The DoS issue

Category: Denial of Service Vulnerability

Content

There is no limit to the depth of the for loop in the multiCall function. If too many proposals are executed in batches, it may cause too much Gas consumption and lead to DoS risks.

Code location: nexvault-wallet-contract/contracts/MultiSigWalletImplementation.sol #L541-555

```
function multiCall(
    Call[] memory calls
) public onlyWallet {
    for (uint i = 0; i < calls.length; i++) {
        if (external_call(
            calls[i].target,
            calls[i].value,
            calls[i].data.length,
            calls[i].data
        )) {}
        else {
            revert("internal call failed");
        }
    }
}
```

Solution

It is recommended to process in batches during the for loop or limit the number of for loops to avoid DoS caused by a large number of loops.

Status

Confirmed

[N2] [Suggestion] The reminder to use the factory contract

Category: Others

Content

The template contract address passed in when creating a multi-signature wallet contract through the factory contract is externally controllable. It should be noted that the external template contract that has not passed the audit cannot be completely trusted.

Solution

It is recommended to pass in the audited external template contract address when creating a multi-signature contract through the factory contract.

Status

Confirmed

[N3] [Suggestion] Batch execution risk

Category: Others

Content

The multiCall function in the MultiSigWalletImplementation contract allows batch execution of proposals. When there is an attacker in the owner list, the attacker can submit a batch of malicious proposals (such as a proposal to delete the owner in batches). Wrong signatures of roles in the owner list will result in a transaction. Execute multiple malicious proposals.Combining this risk point with the attack scenarios of N3 and N4 will cause more serious consequences.

Code location: nexvault-wallet-contract/contracts/MultiSigWalletImplementation.sol #L541-555

```
function multiCall(
    Call[] memory calls
) public onlyWallet {
    for (uint i = 0; i < calls.length; i++) {
        if (external_call(
            calls[i].target,
            calls[i].value,
            calls[i].data.length,
            calls[i].data
        )) {}
        else {
            revert("internal call failed");
        }
    }
}
```

Solution

It is recommended that only a single proposal be allowed to be implemented to prevent serious consequences caused by missigning. Or add restrictions, only allow calls to external contracts, not functions in this contract.

Status

Confirmed

[N4] [Suggestion] Unchecked return value

Category: Others

Content

The external_call function in the MultiSigWalletImplementation contract does not check the return value of the external function when making an external call, and cannot respond in time when the external function fails to execute and returns False but the execution result of the call function returns True.

Code location: nexvault-wallet-contract/contracts/MultiSigWalletImplementation.sol #L301-324

```
    function external_call(
        address destination,
        uint value,
        uint dataLength,
        bytes memory data
    ) internal returns (bool) {
        bool result;
        assembly {
            let x := mload(0x40) // "Allocate" memory for output (0x40 is where "free
memory" pointer is stored by convention)
            let d := add(data, 32) // First 32 bytes are the padded length of data,
so exclude that
            result := call(
                sub(gas(), 34710), // 34710 is the value that solidity is currently
emitting
                // It includes callGas (700) + callVeryLow (3, to pay for SUB) +
callValueTransferGas (9000) +
                // callNewAccountGas (25000, in case the destination address does not exist and needs creating)
                destination,
                value,
                d,
                dataLength, // Size of the input (in bytes) - this is what fixes the padding problem
                x,
                0 // Output is ignored, therefore the output size is zero
            )
        }
        return result;
    }
```

Solution

It is recommended to follow the coding standards and check the return value of the externally called function.

Status

Confirmed

[N5] [High] Risk of front-running attack

Category: Reordering Vulnerability

Content

Users can submit proposals through the batchSignature function in the MultiSigWalletImplementation contract, where the transactionId of the proposal is taken from the transaction.nonce passed in from outside. If there is an attacker in the Owner, the attacker knows the transaction.nonce of the current proposal in advance, and uses this value to create a malicious proposal in advance, which will cause the real proposal to fail transaction [transactionId].destination == address(0) Judging to execute the _addTransaction logic but directly executing the _confirmTransaction logic, which will lead to malicious proposals being executed.

Code location: nexvault-wallet-contract/contracts/MultiSigWalletImplementation.sol #L507-533

```
    function batchSignature(
        Transaction memory transaction,
        Signature[] memory signatureList
    ) public returns (bool isOK) {
        uint transactionId = transaction.nonce;
        for (uint i = 0; i < signatureList.length; i++) {
            Signature memory signature = signatureList[i];
            address signer = signature.signer;
            require(isVerify(transaction, signature));
            if (isOwner[signer]) {
                // addTx
                if (transactions[transactionId].destination == address(0)) {
                    _addTransaction(
                        transaction.nonce,
                        transaction.destination,
                        transaction.value,
                        transaction.data
                    );
                }
                // confirm
                _confirmTransaction(transactionId, signer);
            }
        }
        // execute
        _executeTransaction(transactionId);
        return true;
    }
```

Solution

It is recommended to add verification to the proposal content, and use the contract counter transactionCount to obtain the transactionId instead of customizing it through external input.

Status

Fixed;

[N6] [Suggestion] Parameter type error

Category: Others

Content

In the batchSignature function, the temporary variable txn is used to participate in the Confirmation event record. When the function is executed, the txn variable will be destroyed, which will make it impossible to query the details of the proposal through txId on the chain.

Code location: nexvault-wallet-contract/contracts/MultiSigWalletImplementation.sol #L464-503

```
    function batchSignature(Transaction memory txn, Signature[] memory
sortedSignatures) public returns (bool isOK) {
        require(sortedSignatures.length >= required, "invalid signature data
length");
```

```
        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                DOMAIN_SEPARATOR,
                hashTransaction(txn)
            )
        );
        require(!txNonces[digest], "tx-executed");

        uint256 txId = txn.nonce;
        address lastOwner = address(0);
        for(uint i = 0; i < sortedSignatures.length; i++) {
            Signature memory signature = sortedSignatures[i];
            address signer = signature.signer;
            uint8 v = signature.v;
            bytes32 r = signature.r;
            bytes32 s = signature.s;

            address currentOwner = ecrecover(digest, v, r, s);

            // to save gas, must need signature.signer sorted
            require(currentOwner > lastOwner && isOwner[currentOwner] && signer ==
currentOwner, "error-sig");
            lastOwner = currentOwner;
            emit Confirmation(signer, txId);
        }

        txn.executed = true;
        txNonces[digest] = true;
        if (external_call(txn.destination, txn.value, txn.data.length, txn.data)) {
            emit Execution(txId);
        } else {
            emit ExecutionFailure(txId);
            txn.executed = false;
        }

        return txn.executed;
    }
```

Solution

It is recommended to use global variables to record transaction details.

Status

Confirmed; After communication, the project team will record the details of the proposal under the chain.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002304190002	SlowMist Security Team	2023.04.14 - 2023.04.19	Passed

Summary conclusion: The SlowMist security team used a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 1 high risk and 5 suggestions. The risk of N5 has been fixed. All the other findings have been confirmed. The code has been deployed to the main network and has been open-sourced.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

SLOWMIST

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist