Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

2 Audit Methodology

3 Project Overview

3.1 Project Introduction

3.2 Vulnerability Information

4 Code Overview

4.1 Contracts Description

4.2 Visibility Description

4.3 Vulnerability Summary

5 Audit Result

6 Statement

1 Executive Summary

On 2025.06.16, the SlowMist security team received the edgeX team's security audit application for edgeX, 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 project is a perpetual trading platform built on the StarkWare scaling solution. It realizes three-signature fund custody through the MultiSigPoolV5WithPermit contract (supporting ERC20 exchanges, licensed deposits, and fast withdrawals). By leveraging the modular perpetual contract engine of the StarkPerpetual contract (including position management, validator scheduling, and state storage), it constructs a hybrid architecture of Layer1 asset custody and Layer2 high-frequency trading.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Non-EIP-712 compliant message signing	Others	Low	Acknowledged
N2	Risk of Signature Replay	Replay Vulnerability	Low	Acknowledged
N3	Potential Dos attack in token permit execution	Denial of Service Vulnerability	Low	Acknowledged
N4	External call function check	Others	Suggestion	Acknowledged
N5	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged
N6	Centralization Risk of Signers	Authority Control Vulnerability Audit	Information	Acknowledged

4 Code Overview

4.1 Contracts Description

The main network address of the contract is as follows:

- StarkPerpetual:
  - Proxy: https://etherscan.io/address/0xfaae2946e846133af314d1df13684c89fa7d83dd
  - Impl: https://etherscan.io/address/0x8c43c9bec15d82d153c52518030e0a9590abd35d
- MultiSigPool:
  - https://etherscan.io/address/0xc0a1a1e4af873e9a37a0cac37f3ab81152432cc5
  - https://bscscan.com/address/0x0520b0a951658db92b8a2dd9f146bb8223638740
  - https://arbiscan.io/address/0xceeed84620e5eb9ab1d6dfc316867d2cda332e41

4.2 Visibility Description

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

Proxy			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ProxyRoles
setUpgradeActivationDelay	Private	Can Modify State	-
getUpgradeActivationDelay	Public	-	-
getEnableWindowDuration	Public	-	-
setEnableWindowDuration	Private	Can Modify State	-
implementation	Public	-	-
implementationIsFrozen	Private	Can Modify State	-
initialize	External	-	-
<Receive Ether>	External	Payable	-
<Fallback>	External	Payable	-
setImplementation	Private	Can Modify State	-
isNotFinalized	Public	-	-
setFinalizedFlag	Private	Can Modify State	-
addImplementation	External	Can Modify State	onlyUpgradeGovernor
removeImplementation	External	Can Modify State	onlyUpgradeGovernor
upgradeTo	External	Payable	onlyUpgradeGovernor notFinalized notFrozen

StarkPerpetual			
Function Name	Visibility	Mutability	Modifiers
getNumSubcontracts	Internal	-	-

StarkPerpetual			
magicSalt	Internal	-	-
handlerMapSection	Internal	-	-
expectedIdByIndex	Internal	-	-
initializationSentinel	Internal	-	-

MainDispatcher			
Function Name	Visibility	Mutability	Modifiers
magicSalt	Internal	-	-
handlerMapSection	Internal	-	-
expectedIdByIndex	Internal	-	-
validateSubContractIndex	Internal	-	-
handlingContractId	External	-	-
getSubContractIndex	Internal	-	-
getSubContract	Public	-	-
setSubContractAddress	Internal	Can Modify State	-

MainDispatcherBase			
Function Name	Visibility	Mutability	Modifiers
<Receive Ether>	External	Payable	-
<Fallback>	External	Payable	-
initialize	External	Can Modify State	notCalledDirectly
callExternalInitializer	Private	Can Modify State	-

BlockDirectCall			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Internal	Can Modify State	-

Governance			
Function Name	Visibility	Mutability	Modifiers
getGovernanceInfo	Internal	-	-
initGovernance	Internal	Can Modify State	-
_isGovernor	Internal	-	-
_cancelNomination	Internal	Can Modify State	onlyGovernance
_nominateNewGovernor	Internal	Can Modify State	onlyGovernance
acceptNewGovernor	Private	Can Modify State	-
_acceptGovernance	Internal	Can Modify State	-
_removeGovernor	Internal	Can Modify State	onlyGovernance

MultiSigPoolV5WithPermit			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
<Receive Ether>	External	Payable	-
deposit	Public	Payable	nonReentrant
depositWithPermit	Public	Can Modify State	nonReentrant
withdrawETH	Public	Can Modify State	nonReentrant
withdrawErc20	Public	Can Modify State	nonReentrant
withdrawErc20ForMPC	Public	Can Modify State	nonReentrant

MultiSigPoolV5WithPermit			
factTransferErc20	Public	Can Modify State	nonReentrant
isNative	Internal	-	-
isAllowedSigner	Public	-	-
tryInsertOrderId	Internal	Can Modify State	-
calcSigHash	Public	-	-

4.3 Vulnerability Summary

[N1] [Low] Non-EIP-712 compliant message signing

Category: Others

Content

In the MultiSigPoolV5WithPermit contract, the depositWithPermit, withdrawETH, withdrawErc20, withdrawErc20ForMPC, and factTransferErc20 functions use methods that do not conform to the EIP-712 specification for message signing. In the current implementation, the message data is hashed directly, and then the Ethereum-signed message hash is applied, instead of using the structured data hashing defined in EIP-712.

This approach does not comply with the regulations regarding structured data hashing in the EIP-712 standard. As a result, when users are asked to sign a message, they will see a 32-byte encrypted value instead of a human-readable structured message. This lack of clarity may lead to user confusion and pose potential security risks. Users may sign the data without fully understanding its content or purpose.

Additionally, some mainstream wallets may require users to manually enable the eth_sign functionality to perform such signatures, or they might have disabled eth_sign signatures altogether as a security measure. This requirement creates an extra barrier and significantly degrades the user experience.

Code Location:

./contracts/core/MultiSigPoolV5WithPermit.sol

```
    function depositWithPermit(
      ...
   ) public nonReentrant returns (uint256) {
```

```
    ...



require(ECDSA.recover(ECDSA.toEthSignedMessageHash(keccak256(abi.encodePacked(amount,
starkKey, positionId, block.chainid))), mpcSignature)  == owner,"invalid mpc
signature");

    ...
  }

 function withdrawETH(
    ...
  ) public nonReentrant {
    ...

    bytes32 operationHash = keccak256(abi.encodePacked("ETHER", to, amount,
expireTime, orderId, address(this), block.chainid));
    operationHash = ECDSA.toEthSignedMessageHash(operationHash);

    ...
  }

 function withdrawErc20(
    ...
  ) public nonReentrant {
    ...

    bytes32 operationHash = keccak256(abi.encodePacked("ERC20", to, amount, token,
expireTime, orderId, address(this), block.chainid));
    operationHash = ECDSA.toEthSignedMessageHash(operationHash);

    ...
  }

 function withdrawErc20ForMPC(
    ...
  ) public nonReentrant {
    ...

    bytes32 userOperationHash = keccak256(abi.encodePacked(to, amount, token,
userSignTime, block.chainid));
    userOperationHash = ECDSA.toEthSignedMessageHash(userOperationHash);
    require(ECDSA.recover(userOperationHash, fromUserSignature)  == fromUser,"invalid
from user signature");

    // check withdraw signature
    bytes32 operationHash = keccak256(abi.encodePacked("ERC20", to, amount, token,
expireTime, orderId, address(this), block.chainid,fromUser));
```

```
    operationHash = ECDSA.toEthSignedMessageHash(operationHash);

    ...
  }

  function factTransferErc20(
    ...
  ) public nonReentrant {
    ...

    bytes32 operationHash = keccak256(abi.encodePacked("FAST",to, amount, token,
expireTime, salt, orderId, address(this), block.chainid));
    operationHash = ECDSA.toEthSignedMessageHash(operationHash);

    ...
  }
```

Solution

It is recommended to implement EIP-712 compliant structured data signing.

Status

Acknowledged

[N2] [Low] Risk of Signature Replay

Category: Replay Vulnerability

Content

In the depositWithPermit and withdrawErc20ForMPC functions of the MultiSigPoolV5WithPermit contract, it checks whether the data signature is from an MPC user to conduct deposits and withdrawals for MPC users. However, after verifying the signature, the incoming signature is not marked as used. This means that other users can reuse this signature to call these two functions. This may unexpectedly consume the tokens of legitimate MPC users. For example, the signature can be reused to call the depositWithPermit function.

Code Location:

./contracts/core/MultiSigPoolV5WithPermit.sol

```
  function depositWithPermit(
    ...
  ) public nonReentrant returns (uint256) {
    ...
```

```
require(ECDSA.recover(ECDSA.toEthSignedMessageHash(keccak256(abi.encodePacked(amount,
starkKey, positionId, block.chainid))), mpcSignature)  == owner,"invalid mpc
signature");

    ...
  }

 function withdrawErc20ForMPC(
    ...
  ) public nonReentrant {
    ...

    bytes32 userOperationHash = keccak256(abi.encodePacked(to, amount, token,
userSignTime, block.chainid));
    userOperationHash = ECDSA.toEthSignedMessageHash(userOperationHash);
    require(ECDSA.recover(userOperationHash, fromUserSignature)  == fromUser,"invalid
from user signature");

    ...
  }
```

Solution

It is recommended to check whether the signature has been used in the function, or add an incrementable nonce to the signed data for verification.

Status

Acknowledged; The project team responded: In fact, the mpc signature is used as an anti-money laundering mark, and the deposits and withdrawals in the contract are mainly called by ourselves.

[N3] [Low] Potential Dos attack in token permit execution

Category: Denial of Service Vulnerability

Content

In the MultiSigPoolV5WithPermit contract, users execute account authorization operations and deposit for MPC users by calling the depositWithPermit function. This function will call the permit function of the ERC20 token contract to grant a spending limit. However, if the permit function is preemptively executed by a malicious user (attackers can obtain the corresponding parameters by monitoring the mempool), it may roll back, causing the entire transaction to fail.

Code Location:

./contracts/core/MultiSigPoolV5WithPermit.sol

```
function depositWithPermit(
    ...
  ) public nonReentrant returns (uint256) {
    ...

    // permit call
    IERC20Permit(USDT_ADDRESS).permit(owner,address(this),amount,deadline,v,r,s);

    ...
  }
```

Solution

It is recommended to wrap the permit function call with a try-catch block or implement conditional checks to ensure that if a permit call in a for-loop fails, it does not cause the entire transaction to roll back.

Status

Acknowledged

[N4] [Suggestion] External call function check

Category: Others

Content

In the MultiSigPoolV5WithPermit contract, the deposit function enables users to deposit USDT into the current contract or StarkEx. If the token transferred by the user is not USDT, the AggregationRouter will first be used to exchange the transferred token into USDT. Although the parameters for the exchange are checked here, the function selector for the external call is not examined. If the function selector to be called does not meet expectations, unexpected errors may occur.

Code Location:

./contracts/core/MultiSigPoolV5WithPermit.sol

```
function deposit(
    IERC20 token,
    uint256 amount,
    uint256 starkKey,
```

```
    uint256 positionId,
    bytes calldata exchangeData
  ) public payable nonReentrant returns (uint256) {
    ...

    if (address(token) == USDT_ADDRESS){   // deposit USDT
      ...
    } else {
      (, IAggregationRouterV5.SwapDescription memory desc, ,) =
abi.decode(exchangeData[4:], (address, IAggregationRouterV5.SwapDescription, bytes,
bytes));

      ...

      // Swap token
      (bool success, bytes memory returndata)=
AGGREGATION_ROUTER_V5_ADDRESS.call{value:msg.value}(exchangeData);
      require(success, "exchange failed");

      ...
    }

    ...
  }
```

Solution

It is recommended to also check the function selectors of the externally-called contracts to ensure they meet expectations.

Status

Acknowledged

[N5] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

In the Proxy contract, The UpgradeGovernor role can upgrade the underlying implementation contract. If this role is set to an EOA address and its permission is compromised, it may lead to the underlying implementation contract being upgraded to a malicious one, thus affecting the normal operation of the project.

Code Location:

./starkware/solidity/upgrade/Proxy.sol

```
function upgradeTo(
        address newImplementation,
        bytes calldata data,
        bool finalize
    ) external payable onlyUpgradeGovernor notFinalized notFrozen {
        ...
    }
```

Solution

In the short term, transferring the ownership of core roles to multisig contracts is an effective solution to avoid single-point risk. But in the long run, it is a more reasonable solution to implement a privilege separation strategy and set up multiple privileged roles to manage each privileged function separately. Permissions involving user funds and contract updates should be managed by the community, while permissions involving emergency contract suspensions can be managed by the EOA address. This allows for rapid response to threats while ensuring the safety of user funds.

Status

Acknowledged; The project team responded: The core role of the proxy contract will be replaced from the eoa address to a multi-signature contract.

[N6] [Information] Centralization Risk of Signers

Category: Authority Control Vulnerability Audit

Content

In the MultiSigPoolV5WithPermit contract, when users withdraw funds, they need data signed by two different signers to pass the verification. However, there is a possibility that an attacker, through centralized malicious means, steals the permissions of the two signers and constructs unexpected withdrawal data for signature verification, thereby stealing the funds in the contract.

Code Location:

./contracts/core/MultiSigPoolV5WithPermit.sol

```
function withdrawETH(
    ...
  ) public nonReentrant {
    ...
```

```
      for (uint8 index = 0; index < allSigners.length; index++) {
        address signer = ECDSA.recover(operationHash, signatures[index]);
        require(signer == allSigners[index], "invalid signer");
        require(isAllowedSigner(signer), "not allowed signer");
      }

      ...
    }

     function withdrawErc20(
      ...
    ) public nonReentrant {
      ...

      for (uint8 index = 0; index < allSigners.length; index++) {
        address signer = ECDSA.recover(operationHash, signatures[index]);
        require(signer == allSigners[index], "invalid signer");
        require(isAllowedSigner(signer),"not allowed signer");
      }

      ...
    }

  function factTransferErc20(
      ...
    ) public nonReentrant {
      ...

      for (uint8 index = 0; index < allSigners.length; index++) {
        address signer = ECDSA.recover(operationHash, signatures[index]);
        require(signer == allSigners[index], "invalid signer");
        require(isAllowedSigner(signer),"not allowed signer");
      }

      ...
    }
```

Solution

N/A

Status

Acknowledged

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002506180002	SlowMist Security Team	2025.06.16 - 2025.06.18	Medium Risk

Summary conclusion: The SlowMist security team use a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 1 medium, 3 low risks, 1 suggestion and 1 information. All the findings were acknowledged. The code has been deployed to the mainnet. Since the transfer management of the permissions of the core roles in some contracts of the project has not been carried out yet, the risk level of this report is temporarily medium.

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.

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github
https://github.com/slowmist