Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2026.01.30, the SlowMist security team received the Sunperp Dex team's security audit application for SunperpVaultV2, 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 audit mainly focuses on the on-chain asset vault contract named SunperpVaultV2, which aims to manage native tokens and ERC20 tokens securely. Users can deposit tokens supported by the protocol into the contract, but withdrawals are not initiated directly by users. Instead, they are executed by the operator role. The key point is that each withdrawal must be accompanied by the cryptographic signatures of a pre-set group of validators. The contract will verify the validity of these signatures and whether the total power of the signers reaches the threshold, thus implementing multi-signature authorization to prevent single-point failures or malicious operations.

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	Potential DoS risk	Gas Optimization Audit	Low	Acknowledged
N3	Missing pause status check when making a deposit	Others	Suggestion	Fixed
N4	Risk of Excessive Authority	Authority Control Vulnerability Audit	Medium	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/jerkoyz/sunperp-dex

commit: 70a5b7a56fccf06a5e1b516678eeabed3332d7a4

Fixed Version:

https://github.com/jerkoyz/sunperp-dex

commit: b2c7827b7bb48e6ccba9219b5506ccaee6adfb85

Audit Scope:

```
./contracts
└── SunperpVaultV2.sol
```

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

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

SunperpVaultV2			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
<Receive Ether>	External	Payable	-
initialize	Public	Can Modify State	initializer
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	onlyRole
stop	External	Can Modify State	onlyRole
unstop	External	Can Modify State	onlyRole
_authorizeUpgrade	Internal	Can Modify State	onlyUpgrader
setUpgrader	External	Can Modify State	onlyRole
addValidator	External	Can Modify State	onlyRole
removeValidator	External	Can Modify State	onlyRole
setWhitelist	External	Can Modify State	onlyRole
addToken	External	Can Modify State	onlyRole
removeToken	External	Can Modify State	onlyRole
_transfer	Private	Can Modify State	-
deposit	External	Payable	-
withdrawWhitelist	External	Can Modify State	whenNotPaused whenNotStopped onlyRole

SunperpVaultV2			
withdrawWhitelistV2	External	Can Modify State	whenNotPaused whenNotStopped onlyRole
withdraw	External	Can Modify State	whenNotPaused whenNotStopped onlyRole
withdrawV2	External	Can Modify State	whenNotPaused whenNotStopped onlyRole
_withdraw	Internal	Can Modify State	-
_withdrawV2	Internal	Can Modify State	-
_supportCurrency	Private	-	-
balance	External	-	-
verifyValidatorSignature	Internal	-	-
checkLimit	Internal	Can Modify State	-

4.3 Vulnerability Summary

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

Category: Others

Content

In the SunperpVaultV2 contract, the verifyValidatorSignature function uses a non-EIP-712 compliant method for message signature verification. 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 validators 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 can cause confusion for validators and poses a potential security risk. Validators 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/SunperpVaultV2.sol

```
    function _withdraw(uint256 id, ValidatorInfo[] calldata validators, WithdrawAction
calldata action, bytes[] calldata validatorSignatures) internal {
        ...

        bytes32 digest = keccak256(abi.encode(
            id,
            block.chainid,
            address(this),
            action.token,
            action.receiver,
            action.amount,
            action.deadline
        ));
        verifyValidatorSignature(validators, digest, validatorSignatures);

        ...
    }

    function _withdrawV2(uint256 id, ValidatorInfo[] calldata validators,
WithdrawActionV2 calldata action, bytes[] calldata validatorSignatures) internal {
        ...

        bytes32 digest = keccak256(abi.encode(
            id,
            block.chainid,
            address(this),
            action.token,
            action.owner,
            action.receiver,
            action.amount,
            action.deadline
        ));
        verifyValidatorSignature(validators, digest, validatorSignatures);

        ...
    }

    function verifyValidatorSignature(ValidatorInfo[] calldata validators, bytes32
digest, bytes[] calldata validatorSignatures) internal view {
        ...
```

```
        bytes32 validatorDigest = MessageHashUtils.toEthSignedMessageHash(digest);
        for (uint i = 0; i < validatorSignatures.length && validatorIndex <
validators.length; i ++) {
            address recover = validatorDigest.recover(validatorSignatures[i]);

            ...
        }
        require(power >= totalPower * 3 / 3, "validator signature illegal");
    }
```

Solution

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

Status

Acknowledged;

The project team responded that the reason for using EIP-191 instead of EIP-712 in the SunperpVault contract is that the transaction scenario involves multiple off-chain signature machines handling signatures. There is no need to achieve the visual and readable aspects of EIP-712, and the EIP-191 signature itself can ensure the reliability and legality of the signature.

[N2] [Low] Potential DoS risk

Category: Gas Optimization Audit

Content

In the SunperpVaultV2 contract, In the verifyValidatorSignature function, the loop logic for signature verification assumes that the order of signatures in the validatorSignatures array is strictly consistent with the order of validators in the validators array. Moreover, there is no check in the function to ensure that the lengths of the validatorSignatures array and the validators array are the same. If the signatures are not provided in the expected order, the internal validatorIndex will not be reset, causing subsequent signatures to fail to find a matching validator. Ultimately, this leads to incomplete power calculation and the failure of the withdrawal transaction. Although this does not result in loss of funds, it causes a Denial-of-Service issue for the withdrawal function, preventing legitimate withdrawal requests from being processed.

Code Location:

contracts/SunperpVaultV2.sol

```
    function verifyValidatorSignature(ValidatorInfo[] calldata validators, bytes32
digest, bytes[] calldata validatorSignatures) internal view {
        bytes32 validatorHash = keccak256(abi.encode(validators));
        uint totalPower = availableValidators[validatorHash];
        require(totalPower > 0, "validator illegal");
        uint power = 0;
        uint validatorIndex = 0;
        bytes32 validatorDigest = MessageHashUtils.toEthSignedMessageHash(digest);
        for (uint i = 0; i < validatorSignatures.length && validatorIndex <
validators.length; i ++) {
            address recover = validatorDigest.recover(validatorSignatures[i]);
            if (recover == address(0)) {
                continue;
            }
            while (validatorIndex < validators.length) {
                address validator = validators[validatorIndex].signer;
                validatorIndex ++;
                if (validator == recover) {
                    power += validators[validatorIndex - 1].power;
                    break;
                }
            }
        }
        require(power >= totalPower * 3 / 3, "validator signature illegal");
    }
```

Solution

It is recommended to modify the logic of the verifyValidatorSignature function so that it does not rely on the order of signatures. A feasible approach is to traverse the entire validators array for each signature and record which validators have provided signatures to prevent double-counting of signatures from the same validator.

Status

Acknowledged; The project team responded: The reason we didn't check the length consistency between the validatorSignatures array and the validators array is that there may be a requirement for a two-thirds signature scenario in the future. Also, when making a withdrawal, the operate role will pre - construct the validatorSignatures array and the validators array off-chain to ensure the relative order is consistent when passing parameters on-chain.

[N3] [Suggestion] Missing pause status check when making a deposit

Category: Others

Content

In the SunperpVaultV2 contract, users can transfer supported tokens into the contract by calling the deposit function. However, this function lacks a check on whether the contract is currently paused. If the contract is paused and withdrawals are not possible, the tokens transferred in by users may not be retrievable.

Code Location:

contracts/SunperpVaultV2.sol

```
    function deposit(address currency, uint256 amount, uint256 broker) external
payable {
        ...
    }
```

Solution

It is recommended to add the whenNotPaused and whenNotStopped modifier checks to the deposit function as well.

Status

Fixed

[N4] [Medium] Risk of Excessive Authority

Category: Authority Control Vulnerability Audit

Content

In the SunperpVaultV2 contract, the ADMIN_ROLE can change the upgrader address of the contract by calling the setUpgrader function, and the upgrader address has the right to upgrade the contract to any new implementation. Additionally, the ADMIN_ROLE can call the addValidator function to add any signature validator to the contract. The data during withdrawal needs to be verified through the signatures of these validators.

If these roles are set to EOA addresses and their private keys are leaked or stolen, it may affect the fund security within the contract and the normal operation of its functions.

Code Location:

contracts/SunperpVaultV2.sol

```
    function _authorizeUpgrade(address newImplementation) internal onlyUpgrader
override {}

    function setUpgrader(address newUpgrader) external onlyRole(ADMIN_ROLE) {
```

```
        ...
    }

    function addValidator(ValidatorInfo[] calldata validators) external
onlyRole(ADMIN_ROLE) {
        ...
    }
```

Solution

In the short term, transferring the ownership of core roles to time-locked contracts and managing them by multi-signatures is an effective solution to avoid single-point risks. However, in the long run, a more reasonable solution is to implement a permission separation strategy and set up multiple privileged roles to manage each privileged function separately. Permissions involving user funds and contract core parameter updates should be managed by the community, while permissions involving emergency contract suspensions can be managed by EOA addresses. This ensures the safety of user funds while responding quickly to threats.

Status

Acknowledged; The project team responded that the ADMIN_ROLE indeed has excessive permissions, and the configured address is an EOA address. However, since our contract is deployed on the TRON blockchain, TRON EOA accounts can be managed through multi-signature. We plan to assign a multi-signature-managed EOA address to the ADMIN_ROLE. This relatively avoids the situation where the leakage of the EOA private key would affect the normal functions of the contract.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002601300002	SlowMist Security Team	2026.01.30 - 2026.01.30	Medium Risk

Summary conclusion: The SlowMist security team uses a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 1 medium, 2 low risk and 1 suggestion. All the findings were fixed and acknowledged. Since the project has not yet been deployed to the mainnet and the permission of the core role has not yet been transferred, the risk level reported 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