SSI Protocol

Smart Contract Security Audit Report

Audited by: TenArmor
Audit dates: Dec 02 - Dec 06

Content

1. Introduction

1.1 About TenArmor

TenArmor is a leading Web3 security firm dedicated to safeguarding blockchain ecosystems. We offer cutting-edge security solutions tailored to the complexities of decentralized technologies. With our flagship products, ArgusAlert for real-time threat detection and VulcanShield for proactive attack interception, we provide comprehensive protection against evolving threats. Our expertise spans smart contract auditing, cryptocurrency tracing, and beyond, making TenArmor a trusted partner for organizations aiming to secure their digital assets in the decentralized world.
Learn more about us:
Official Website: TenArmor.com
X: @TenArmor, @TenArmorAlert
E-mail: team@tenarmor.com

1.2 Disclaimer

This report represents an analysis performed within a defined scope and time frame, based on the materials and documentation provided. It is not exhaustive and does not encompass all potential vulnerabilities.

A smart contract security review aims to identify as many vulnerabilities as possible within the given constraints of time, resources, and expertise. However, it cannot guarantee the complete absence of vulnerabilities or the complete security of the project. This report does not serve as an endorsement of any specific project or team.

To enhance security, we strongly recommend conducting subsequent security reviews, implementing bug bounty programs, and maintaining ongoing on-chain monitoring.

1.3 Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About SSI Protocol

The SSI Protocol utilizes smart contracts to consolidate multi-chain and multi-asset portfolios into a single Wrapped Token. Each Wrapped Token is a representation of a

diverse collection of underlying assets, designed to mirror the value changes of the entire portfolio. This approach streamlines asset management and enables seamless tracking of complex asset baskets.

2.2 Audit Scope

Repository: https://github.com/SoSoValueLabs/ssi-protocol
Audit Commit Hash: https://github.com/SoSoValueLabs/ssi-protocol/commit/7929bfe83397e5f6f3dcacc52eaa94b762073ecf
Fixes Commit Hash: https://github.com/SoSoValueLabs/ssi-protocol/tree/7e3adbfc8f4ef3dbc08be151341314aaad687d40

2.3 Findings Count

Severity	Count
High	1
Medium	2
Low	1
Informational	2
Total Findings	**6**

3. Findings Summary

ID	Description	Status
[H-1]	Incorrect check for amount in function withdraw()	Resolved
[M-1]	Missing update values in the mappings within functions setIssuer()/setRebalancer()/setFeeManager()	Resolved
[M-2]	Missing implementation of pause() and unpause() in contract USSI	Resolved
[L-1]	Missing check for chainId in Order and HedgeOrder	Resolved
[I-1]	Update state before token transfer to avoid potential reentrancy	Resolved
[I-2]	Use safeTransfer/safeTransferFrom	Resolved

4. Findings

4.1 High Risk

[H-1]	Incorrect check for amount in function withdraw()	Resolved

Description

In the AssetLocking contract, the withdraw() function performs a check lockData.amount <= lockData.cooldownAmount to ensure there are sufficient funds available for withdrawal. However, lockData.amount actually refers to the locked tokens, which are not eligible for withdrawal. Therefore, this check is flawed and could cause users to experience issues when attempting to withdraw their funds.

src/AssetLocking.sol Line#120

```
120     function withdraw(address token, uint256 amount) external whenNotPaused {
121         LockData storage lockData = lockDatas[token][msg.sender];
122         require(lockData.cooldownAmount > 0, "nothing to withdraw");
123         require(lockData.cooldownEndTimestamp <= block.timestamp, "coolingdown");
124         require(lockData.amount <= lockData.cooldownAmount, "no enough balance to withdraw");
125         IERC20(token).safeTransfer(msg.sender, amount);
126         lockData.cooldownAmount -= amount;
127         LockConfig storage lockConfig = lockConfigs[token];
128         lockConfig.totalCooldown -= amount;
129         emit Withdraw(msg.sender, token, amount);
130     }
131 }
```

Impact

It could cause users to experience issues when attempting to withdraw their funds.

Recommendation

Use the check amount <= lockData.cooldownAmount.

SSI: Fixed

4.2 Medium Risk

[M-1]	Missing update mapping values in functions setIssuer()/setRebalancer()/setFeeManager()	Resolved

Description

The AssetFactory.setIssuer() function is designed to replace the current issuer by performing two actions:

- Revoking the role from the old issuer
- Granting the role to the new issuer

However, the method currently fails to update the issuers[assetID], leaving the issuer information outdated.

The functions setRebalancer() and setFeeManager() have similar issues.

src/AssetFactory.sol Line#100

```
100     function setIssuer(uint256 assetID, address issuer) external onlyOwner {
101         require(issuer != address(0), "issuer is zero address");
102         require(assetIDs.contains(assetID), "assetID not exists");
103         IAssetToken assetToken = IAssetToken(assetTokens[assetID]);
104         require(!assetToken.issuing(), "is issuing");
105         address oldIssuer = issuers[assetID];
106         assetToken.revokeRole(assetToken.ISSUER_ROLE(), oldIssuer);
107         assetToken.grantRole(assetToken.ISSUER_ROLE(), issuer);
108         emit SetIssuer(assetID, oldIssuer, issuer);
109     }
110
111     function setRebalancer(uint256 assetID, address rebalancer) external onlyOwner {
112         require(rebalancer != address(0), "rebalancer is zero address");
113         require(assetIDs.contains(assetID), "assetID not exists");
114         IAssetToken assetToken = IAssetToken(assetTokens[assetID]);
115         require(!assetToken.rebalancing(), "is rebalancing");
116         address oldRebalancer = rebalancers[assetID];
117         assetToken.revokeRole(assetToken.REBALANCER_ROLE(), oldRebalancer);
118         assetToken.grantRole(assetToken.REBALANCER_ROLE(), rebalancer);
119         emit SetRebalancer(assetID, oldRebalancer, rebalancer);
120     }
121
122     function setFeeManager(uint256 assetID, address feeManager) external onlyOwner {
123         require(feeManager != address(0), "feeManager is zero address");
124         require(assetIDs.contains(assetID), "assetID not exists");
125         IAssetToken assetToken = IAssetToken(assetTokens[assetID]);
126         address oldFeeManager = feeManagers[assetID];
127         assetToken.revokeRole(assetToken.FEEMANAGER_ROLE(), oldFeeManager);
128         assetToken.grantRole(assetToken.FEEMANAGER_ROLE(), feeManager);
129         emit SetFeeManager(assetID, oldFeeManager, feeManager);
130     }
```

Impact

This results in two issues:

- The previous issuer will remain authorized if it is updated again.
- All calls to AssetFactory.issuers() will fail.

Recommendation

Update the corresponding values in the mappings within these functions.

SSI: Fixed

[M-2]	Missing implementation of pause() and unpause() in contract USSI	Resolved

Description

The USSI contract inherits from the PausableUpgradeable contract, but it does not implement the pause() and unpause() functions. As a result, the pausing and unpausing mechanism will not work as intended.

src/USSI.sol Line#19

```
19  contract USSI is Initializable, OwnableUpgradeable, AccessControlUpgradeable, ERC20Upgradeable, UUPSUpgradeable, PausableUpgradeable {
20      using EnumerableSet for EnumerableSet.Bytes32Set;
21      using EnumerableSet for EnumerableSet.UintSet;
22      using SafeERC20 for IERC20;
23
```

Impact
The pausing and unpausing mechanism will not work as intended.

Recommendation
Implement the pause() and unpause() functions in the contract.

SSI: Fixed

4.3 Low Risk

[L-1]	Missing check for chainId in Order and HedgeOrder	Resolved

Description
In the USSI contract, the struct HedgeOrder lacks a chainId parameter. As a result, if the contract is deployed across multiple chains, a single signature could be replayed on different chains, potentially leading to the unauthorized use of signed HedgeOrders across these chains.
The struct Order has the similar issue.

src/USSI.sol Line#122

```
122     function checkHedgeOrder(HedgeOrder calldata hedgeOrder, bytes32 orderHash, bytes calldata orderSignature) public view {
123         if (hedgeOrder.orderType == HedgeOrderType.MINT) {
124             require(supportAssetIDs.contains(hedgeOrder.assetID), "assetID not supported");
125         }
126         if (hedgeOrder.orderType == HedgeOrderType.REDEEM) {
127             require(redeemToken == hedgeOrder.redeemToken, "redeem token not supported");
128         }
129         require(block.timestamp <= hedgeOrder.deadline, "expired");
130         require(!orderHashs.contains(orderHash), "order already exists");
131         require(SignatureChecker.isValidSignatureNow(orderSigner, orderHash, orderSignature), "signature not valid");
132     }
```

```
27      struct HedgeOrder {
28          HedgeOrderType orderType;
29          uint256 assetID;
30          address redeemToken;
31          uint256 nonce;
32          uint256 inAmount;
33          uint256 outAmount;
34          uint256 deadline;
35          address requester;
36      }
```

Impact
This allows users to reuse a signature intended for one chain and execute it on a different chain.

Recommendation
Add a chainId field to the HedgeOrder and Order structs, and verify the chainId when validating signatures.

SSI: Fixed

4.4 Informational

[I-1]	Update state before token transfer to avoid potential reentrancy	Resolved

Description
In the withdraw() function of the StakeToken contract, cooldownInfo.cooldownAmount is subtracted after the token transfer, which creates an opportunity for reentrancy if the token is untrusted.
The withdraw() function in AssetLocking contract has the similar issue.

src/StakeToken.sol Line#82

```
77      function withdraw(uint256 amount) external whenNotPaused {
78          CooldownInfo storage cooldownInfo = cooldownInfos[msg.sender];
79          require(cooldownInfo.cooldownAmount >= amount, "not enough cooldown amount");
80          require(cooldownInfo.cooldownEndTimestamp <= block.timestamp, "cooldowning");
81          IERC20(token).safeTransfer(msg.sender, amount);
82          cooldownInfo.cooldownAmount -= amount;
83          emit Withdraw(msg.sender, amount);
84      }
85  }
```

Recommendation
To adhere to the CEI (Check-Effect-Interaction) pattern, update the cooldownInfo before transferring the token.

SSI: Fixed

[I-2]	Use safeTransfer/safeTransferFrom	Resolved

Description
The lock() function, which uses IERC20(token).transferFrom(), may fail for certain irregular ERC20 tokens. Some ERC20 tokens have a transferFrom method without a return value, which can cause the lock() function to fail.
The transfer function in rejectMint() and rejectRedeem() functions within the USSI

contract have similar issues.

src/AssetLocking.sol Line#102

```
95      function lock(address token, uint256 amount) external whenNotPaused {
96          require(tokens_.contains(token), "token not supported");
97          require(lockConfigs[token].epoch == activeEpochs[token], "token cannot stake now");
98          LockData storage lockData = lockDatas[token][msg.sender];
99          LockConfig storage lockConfig = lockConfigs[token];
100         require(lockConfig.totalLock + amount <= lockConfig.lockLimit, "total lock amount exceeds lock limit");
101         require(IERC20(token).allowance(msg.sender, address(this)) >= amount, "not enough allowance");
102         IERC20(token).transferFrom(msg.sender, address(this), amount);
103         lockData.amount += amount;
104         lockConfig.totalLock += amount;
105         emit Lock(msg.sender, token, amount);
106     }
```

Recommendation

Use SafeERC20.safeTransferFrom/safeTransfer

SSI: Fixed