Security Audit Report for SSI Protocol

Date: December 18, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	SoSoValueLabs
Target	SSI Protocol

Version History

Version	Date	Description
1.0	December 18, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-12-24 02:28:07+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

This audit focuses on the code repositories of the SSI Protocol [1] of SoSoValueLabs. The SSI Protocol leverages on-chain smart contracts to repackage multi-chain, multi-asset portfolios into Wrapped Tokens. These tokens represent a basket of underlying assets, enabling Wrapped Tokens to track the value fluctuations of the basket.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
SSI Protocol	Version 1	7929bfe83397e5f6f3dcacc52eaa94b762073ecf
	Version 2	4ff5f0db5951905f277d5e5a71025f0968102c06

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] https://github.com/SoSoValueLabs/ssi-protocol

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

Table 1.1: Vulnerability Severity Classification

		Likelihood	
		High	*Low*
Impact	*High*	High	Medium
	Low	Medium	Low

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **five** potential security issues. Besides, we have **five** recommendations and **five** notes.

- High Risk: 2
- Medium Risk: 3
- Recommendation: 5
- Note: 5

ID	Severity	Description	Category	Status
1	High	Incorrect check on amount in function `withdraw()`	DeFi Security	Fixed
2	High	Insufficient status check in function `rejectRedeemRequest()`	DeFi Security	Fixed
3	Medium	Lack of implementation of `pause()` and `unpause()` in contract `USSI`	DeFi Security	Fixed
4	Medium	Potential replay attack in `HedgeOrder` and `OrderInfo`	DeFi Security	Fixed
5	Medium	Potential out-of-gas when processing loops	DeFi Security	Fixed
6	-	Fix the typos	Recommendation	Fixed
7	-	Lack of invoking function `_disableInitializers()`	Recommendation	Fixed
8	-	Remove unnecessary checks	Recommendation	Confirmed
9	-	Check parameters in the constructors and initializers	Recommendation	Fixed
10	-	Use safe ERC-20 operations	Recommendation	Fixed
11	-	Potential centralization risk	Note	-
12	-	Withdrawal may not occur within the expected timeframe	Note	-
13	-	Limited support tokens in the protocol	Note	-
14	-	Inconsistency of participant permissions in contracts AssetIssuer and USSI	Note	-
15	-	Additional checks for rescuing funds	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect check on amount in function `withdraw()`

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description In the contract AssetLocking, the function withdraw() checks lockData.amount <= lockData.cooldownAmount to make sure that there is enough balance to withdraw. However lockData.amount is actually the amount of locked tokens which cannot be withdrawn. Therefore, the check is wrong and may result in failure of fund withdrawals for users.

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

Listing 2.1: src/AssetLocking.sol

Impact It can result in failure of fund withdrawals for users.

Suggestion Change the check to amount <= lockData.cooldownAmount.

2.1.2 Insufficient status check in function rejectRedeemRequest()

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description In the function rejectRedeemRequest(), the status check on swapRequest.status is to check whether swap requests are rejected. However, it does not consider the situation when swap requests are cancelled. Therefore, when a swap request is cancelled, the corresponding redeem request can not be rejected or confirmed, leading to the DoS of the minting and redeeming process for the corresponding asset tokens.

```
function rejectRedeemRequest(uint nonce) external onlyOwner {
    require(nonce < redeemRequests.length, "nonce too large");
    Request memory redeemRequest = redeemRequests[nonce];
    require(redeemRequest.status == RequestStatus.PENDING, "redeem request is not pending");
    ISwap swap = ISwap(redeemRequest.swapAddress);
    SwapRequest memory swapRequest = swap.getSwapRequest(redeemRequest.orderHash);
    require(swapRequest.status == SwapRequestStatus.REJECTED, "swap request is not rejected");
    IAssetToken assetToken = IAssetToken(redeemRequest.assetTokenAddress);
    require(assetToken.balanceOf(address(this)) >= redeemRequest.amount, "not enough asset token to transfer");
    assetToken.safeTransfer(redeemRequest.requester, redeemRequest.amount);
    redeemRequests[nonce].status = RequestStatus.REJECTED;
    assetToken.unlockIssue();
    emit RejectRedeemRequest(nonce);
}
```

Listing 2.2: src/AssetIssuer.sol

Impact This will lead to the malfunction of the corresponding asset token and the contract `AssetIssuer`.

Suggestion Change the check to `swapRequest.status == SwapRequestStatus.REJECTED || swapRequest.status == SwapRequestStatus.CANCEL`.

2.1.3 Lack of implementation of `pause()` and `unpause()` in contract `USSI`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `USSI` inherits from the contract `PausableUpgradeable`. However, it does not implement the functions `pause()` and `unpause()`. This will lead to the result that the mechanism of pausing and unpausing can not function as expected.

```
contract USSI is Initializable, OwnableUpgradeable, AccessControlUpgradeable, ERC20Upgradeable,
      UUPSUpgradeable, PausableUpgradeable {
```

Listing 2.3: src/USSI.sol

Impact The mechanism of pausing and unpausing cannot function as expected.

Suggestion Implement the functions of `pause()` and `unpause()`.

2.1.4 Potential replay attack in `HedgeOrder` and `OrderInfo`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contracts `USSI` and `Swap` lack a field for the corresponding chain in the structs `HedgeOrder` and `OrderInfo`. When deployed on multiple chains, this omission enables the replay of a single signature (corresponding to a single order) across different chains, potentially resulting in the multiple usages of `OrderInfo` and `HedgeOrder` across multiple chains.

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

Listing 2.4: src/USSI.sol

```
function checkOrderInfo(OrderInfo memory orderInfo) public view returns (uint) {
    if (block.timestamp >= orderInfo.order.deadline) {
        return 1;
    }
    bytes32 orderHash = keccak256(abi.encode(orderInfo.order));
    if (orderHash != orderInfo.orderHash) {
        return 2;
    }
    if (!SignatureChecker.isValidSignatureNow(orderInfo.order.maker, orderHash, orderInfo.orderSign)) {
        return 3;
    }
    if (orderHashs.contains(orderHash)) {
        return 4;
    }
    if (orderInfo.order.inAddressList.length != orderInfo.order.inTokenset.length) {
        return 5;
    }
    if (orderInfo.order.outAddressList.length != orderInfo.order.outTokenset.length) {
        return 6;
    }
    if (!hasRole(MAKER_ROLE, orderInfo.order.maker)) {
        return 7;
    }
    for (uint i = 0; i < orderInfo.order.outAddressList.length; i++) {
        if (!outWhiteAddresses[orderInfo.order.outAddressList[i]]) {
            return 8;
        }
    }
    return 0;
}
```

Listing 2.5: src/Swap.sol

Impact This may cause the multiple usages of signed orders on multiple chains.

Suggestion Add a check on the corresponding chain when verifying orders.

2.1.5 Potential out-of-gas when processing loops

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `AssetFactory`, there is no upper limit set for creating asset tokens. As a result, the array `assetIDs` can grow excessively large. This excessive growth poses a risk of causing an out-of-gas error in the function `setTokenImpl()`, which iterates over the entire array `assetIDs`. Such an error prevents the upgrade of existing asset tokens. Similarly, the contract `StakeFactory` suffers from the same issue.

```
function setTokenImpl(address tokenImpl_) external onlyOwner {
    require(tokenImpl_ != address(0), "token impl address is zero");
    require(tokenImpl_ != tokenImpl, "token impl is not change");
    tokenImpl = tokenImpl_;
    emit SetTokenImpl(tokenImpl);
    for (uint i = 0; i < assetIDs.length(); i++) {
        address assetToken = assetTokens[assetIDs.at(i)];
        UUPSUpgradeable(assetToken).upgradeToAndCall(tokenImpl, new bytes(0));
        emit UpgradeAssetToken(assetIDs.at(i), tokenImpl);
    }
}
```

Listing 2.6: src/AssetFactory.sol

```
function _setSTImpl(address stImpl_) internal {
    require(stImpl_ != address(0), "stImpl is zero address");
    require(stImpl_ != stImpl, "stImpl not change");
    for (uint i = 0; i < assetIDs.length(); i++) {
        address stakeToken = stakeTokens[assetIDs.at(i)];
        UUPSUpgradeable(stakeToken).upgradeToAndCall(stImpl_, new bytes(0));
        emit UpgradeStakeToken(assetIDs.at(i), stImpl, stImpl_);
    }
    emit SetSTImpl(stImpl, stImpl_);
    stImpl = stImpl_;
}
```

Listing 2.7: src/StakeFactory.sol

Impact Potential out-of-gas when processing asset upgrades.

Suggestion Add an input parameter of an array of assetIDs in the function `setTokenImpl()`, as well as the function `setSTImpl()`.

2.2 Recommendations

2.2.1 Fix the typos

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Several `require` statements contain typos. For example, in the following code segment, the error message should be “too little left in the new tokenset”.

```
    require(newTokenset[i].amount > 0, "too little left in new basket");
```

Listing 2.8: src/AssetRebalancer.sol

For example, in the following code segment, the error message should be “tokenset length not match addressList length”.

```
    require(tokenset.length == addressList.length, "tokenset length not maatch addressList
        length");
```

Listing 2.9: src/Swap.sol

Suggestion Fix the typos.

2.2.2 Lack of invoking function _disableInitializers()

Status Fixed in Version 2

Introduced by Version 1

Description In the contracts USSI, AssetFactory, AssetLocking, AssetToken, StakeFactory and StakeToken, the function _disableInitializers() is not invoked in the function constructor(). Invoking this function prevents the contract itself from being initialized, thereby avoiding unexpected scenarios.

```
/**
 * @dev Locks the contract, preventing any future reinitialization. This cannot be part of an
     initializer call.
 * Calling this in the constructor of a contract will prevent that contract from being
     initialized or reinitialized
 * to any version. It is recommended to use this to lock implementation contracts that are
     designed to be called
 * through proxies.
 *
 * Emits an {Initialized} event the first time it is successfully executed.
 */
function _disableInitializers() internal virtual {
    // solhint-disable-next-line var-name-mixedcase
    InitializableStorage storage $ = _getInitializableStorage();

    if ($._initializing) {
        revert InvalidInitialization();
    }
    if ($._initialized != type(uint64).max) {
        $._initialized = type(uint64).max;
        emit Initialized(type(uint64).max);
    }
}
```

Listing 2.10: lib/openzeppelin-contracts-upgradeable/contracts/proxy/utils/Initalizable.sol

Suggestion Invoke the function _disableInitializers() in the function constructor().

2.2.3 Remove unnecessary checks

Status Confirmed

Introduced by Version 1

Description There are multiple unnecessary checks in the protocol, which are listed as follows:

1. In the contract `AssetFactory`, there is a check to see if the state variable `assetIDs` contains the `assetId`. However, if the corresponding `assetToken` does not exist, the related function will not be called successfully. Therefore, the checks including but not limited to the following code segments are unnecessary.

```
    require(assetIDs.contains(assetID), "assetID not exists");
    IAssetToken assetToken = IAssetToken(assetTokens[assetID]);
    require(!assetToken.issuing(), "is issuing");
```

Listing 2.11: src/AssetFactory.sol

```
    require(assetIDs.contains(assetID), "assetID not exists");
    IAssetToken assetToken = IAssetToken(assetTokens[assetID]);
    require(!assetToken.rebalancing(), "is rebalancing");
```

Listing 2.12: src/AssetFactory.sol

```
    require(assetIDs.contains(assetID), "assetID not exists");
    IAssetToken assetToken = IAssetToken(assetTokens[assetID]);
    address oldFeeManager = feeManagers[assetID];
    assetToken.revokeRole(assetToken.FEEMANAGER_ROLE(), oldFeeManager);
```

Listing 2.13: src/AssetFactory.sol

2. The function `transferFrom()` will automatically revert if the balance or allowance is insufficient during execution. Thus, the checks including but not limited to the following code segments are unnecessary.

```
    require(assetToken.balanceOf(msg.sender) >= order.inAmount, "not enough asset token
        balance");
    require(assetToken.allowance(msg.sender, address(this)) >= order.inAmount, "not enough
         asset token allowance");
```

Listing 2.14: src/AssetIssuer.sol

```
    require(IERC20(token).allowance(msg.sender, address(this)) >= amount, "not enough
        allowance");
```

Listing 2.15: src/AssetLocking.sol

```
    require(IERC20(token).allowance(msg.sender, address(this)) >= amount, "not enough
        allowance");
```

Listing 2.16: src/StakeToken.sol

```
        require(token.balanceOf(from) >= tokenAmount, "not enough balance");
        require(token.allowance(from, address(this)) >= tokenAmount, "not enough allowance
            ");
```

Listing 2.17: src/Swap.sol

```
    require(assetToken.allowance(hedgeOrder.requester, address(this)) >= hedgeOrder.
        inAmount, "not enough allowance");
```

Listing 2.18: src/USSI.sol

```
    require(allowance(hedgeOrder.requester, address(this)) >= hedgeOrder.inAmount, "not enough allowance");
```

Listing 2.19: src/USSI.sol

3. Afte rSolidity version 0.8.0, if an underflow occurs, the transaction will revert. Thus, the following check is redundant.

```
    require(lockData.amount >= amount, "not enough balance to unlock");
```

Listing 2.20: src/AssetLocking.sol

4. The role validation through the function `hasRole()` on the asset tokens are mostly redundant, as the contract `AssetToken` implements role checking. Therefore, the checks including but not limited to the following are redundant.

```
function setFee(uint256 assetID, uint256 fee) external onlyOwner {
    IAssetFactory factory = IAssetFactory(factoryAddress);
    IAssetToken assetToken = IAssetToken(factory.assetTokens(assetID));
    require(assetToken.feeCollected(), "has fee not collected");
    require(assetToken.hasRole(assetToken.FEEMANAGER_ROLE(), address(this)), "not a fee manager");
    assetToken.setFee(fee);
}

function collectFeeTokenset(uint256 assetID) external onlyOwner {
    IAssetFactory factory = IAssetFactory(factoryAddress);
    IAssetToken assetToken = IAssetToken(factory.assetTokens(assetID));
    require(assetToken.hasRole(assetToken.FEEMANAGER_ROLE(), address(this)), "not a fee manager");
    require(assetToken.rebalancing() == false, "is rebalancing");
    require(assetToken.issuing() == false, "is issuing");
    assetToken.collectFeeTokenset();
}
```

Listing 2.21: src/AssetFeeManager.sol

Suggestion Remove these unnecessary code segments to save gas.

Feedback from the project These validations are used to facilitate debugging by providing correct error messages.

2.2.4 Check parameters in the constructors and initializers

Status Fixed in `Version 2`

Introduced by `Version 1`

Description It is recommended to add sanity checks for parameters in the functions `constructor()` and `initialize()`. For example, in the following code segment, the function `constructor()` does not check whether the parameter `factoryAddress` is zero.

```
constructor(address owner, address factoryAddress_) Ownable(owner) {
    factoryAddress = factoryAddress_;
}
```

Listing 2.22: src/AssetController.sol

In the following code segment, it is not checked whether the addresses `factoryAddress_`, `stImpl_` are zero.

```
function initialize(address owner, address factoryAddress_, address stImpl_) public initializer
     {
    __Ownable_init(owner);
    __UUPSUpgradeable_init();
    factoryAddress = factoryAddress_;
    _setSTImpl(stImpl_);
}
```

Listing 2.23: src/StakeFactory.sol

Suggestion Check parameters in the constructors.

2.2.5 Use safe ERC-20 operations

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contracts `USSI` and `AssetIssuer` should avoid setting approval for other contracts to `type(uint256).max`, as issues with the authorized contracts could lead to significant losses.

```
 if (inToken.allowance(address(this), swapAddress) < inTokenAmount) {
     inToken.forceApprove(swapAddress, type(uint256).max);
 }
```

Listing 2.24: src/AssetIssuer.sol

```
 if (assetToken.allowance(address(this), address(issuer)) < hedgeOrder.inAmount) {
     assetToken.approve(address(issuer), type(uint256).max);
 }
```

Listing 2.25: src/USSI.sol

Use the `SafeERC20` library for ERC-20 operations to ensure the safety of ERC-20 operations.

```
function rejectMint(bytes32 orderHash) external onlyOwner {
    require(orderHashs.contains(orderHash), "order not exists");
    require(orderStatus[orderHash] == HedgeOrderStatus.PENDING, "order is not pending");
    HedgeOrder storage hedgeOrder = hedgeOrders[orderHash];
    require(hedgeOrder.orderType == HedgeOrderType.MINT, "order type not match");
    IERC20 assetToken = IERC20(IAssetFactory(factoryAddress).assetTokens(hedgeOrder.assetID));
    assetToken.transfer(hedgeOrder.requester, hedgeOrder.inAmount);
    orderStatus[orderHash] = HedgeOrderStatus.REJECTED;
    emit RejectMint(orderHash);
}
```

Listing 2.26: src/USSI.sol

```
function rejectRedeem(bytes32 orderHash) external onlyOwner {
    require(orderHashs.contains(orderHash), "order not exists");
    require(orderStatus[orderHash] == HedgeOrderStatus.PENDING, "order is not pending");
    HedgeOrder storage hedgeOrder = hedgeOrders[orderHash];
    require(hedgeOrder.orderType == HedgeOrderType.REDEEM, "order type not match");
    transfer(hedgeOrder.requester, hedgeOrder.inAmount);
    orderStatus[orderHash] = HedgeOrderStatus.REJECTED;
    emit RejectRedeem(orderHash);
}
```

Listing 2.27: src/USSI.sol

Suggestion Use safe ERC-20 operations and apply stricter controls on the usage of approvals.

2.3 Notes

2.3.1 Potential centralization risk

Introduced by Version 1

Description The protocol has several centralization-related issues, which are as follows:

1. We assume that all the roles which are controlled by the protocol maintainers to validate all the inputs and function correctly according to the documentation. Specifically, the following roles are fully trusted:
   (a). Owner and Default Admin.
   (b). Issuer, Fee Manager and Rebalancer.
   (c). Takers and Makers for the contract Swap.
   (d). Participants.
2. Function AssetIssuer.withdraw() is used to rescue the tokens that are transferred in by mistake. However, according to the current implementation, the contract's owner can withdraw all the funds from the contract if there is no assetToken in issuing state. In this case, if the owner's private key is compromised or lost, it could lead to losses for the users.

```
function withdraw(address[] memory tokenAddresses) external onlyOwner {
    IAssetFactory factory = IAssetFactory(factoryAddress);
    uint256[] memory assetIDs = factory.getAssetIDs();
    for (uint i = 0; i < assetIDs.length; i++) {
        IAssetToken assetToken = IAssetToken(factory.assetTokens(assetIDs[i]));
        require(!assetToken.issuing(), "is issuing");
    }
    for (uint i = 0; i < tokenAddresses.length; i++) {
        if (tokenAddresses[i] != address(0)) {
            IERC20 token = IERC20(tokenAddresses[i]);
            token.safeTransfer(owner(), token.balanceOf(address(this)));
        }
    }
}
```

Listing 2.28: src/AssetIssuer.sol

3. The contract Swap highly relies on the off-chain verification of the transaction hashes used for the swap. Therefore, it requires both makers and takers are trusted and validate the transaction hashes properly before confirming on the swap requests.

```
function makerConfirmSwapRequest(OrderInfo memory orderInfo, bytes[] memory outTxHashs)
     external onlyRole(MAKER_ROLE) whenNotPaused {
    validateOrderInfo(orderInfo);
    bytes32 orderHash = orderInfo.orderHash;
    SwapRequest memory swapRequest = swapRequests[orderHash];
    require(orderInfo.order.maker == msg.sender, "not order maker");
    require(swapRequest.status == SwapRequestStatus.PENDING, "status error");
    if (swapRequest.outByContract) {
        transferTokenset(msg.sender, orderInfo.order.outTokenset, orderInfo.order.
             outAmount, orderInfo.order.outAddressList);
    } else {
        require(orderInfo.order.outTokenset.length == outTxHashs.length, "wrong outTxHashs
             length");
        swapRequests[orderHash].outTxHashs = outTxHashs;
    }
    swapRequests[orderHash].status = SwapRequestStatus.MAKER_CONFIRMED;
    swapRequests[orderHash].blocknumber = block.number;
    emit MakerConfirmSwapRequest(msg.sender, orderHash);
}
```

Listing 2.29: src/Swap.sol

4. The maker must complete the transfer first, followed by the taker. To securely complete this process, the taker must be a fully trustedwhitelisted role. Malicious takers can potentially cancel the transfer after the makers complete their transaction, causing losses to the makers.
5. When outByContract is false, function rollbackSwapRequest() can change the status of an order from MAKER_CONFIRMED to PENDING. However, all the related funds which makers transferred during the confirmation are not handled on-chain. Therefore, it requires the fully trusted property of the takers.

```
function rollbackSwapRequest(OrderInfo memory orderInfo) external onlyRole(TAKER_ROLE)
     whenNotPaused {
    validateOrderInfo(orderInfo);
    bytes32 orderHash = orderInfo.orderHash;
    require(swapRequests[orderHash].requester == msg.sender, "not order taker");
    require(swapRequests[orderHash].status == SwapRequestStatus.MAKER_CONFIRMED, "swap
         request status is not maker_confirmed");
    require(!swapRequests[orderHash].outByContract, "out by contract cannot rollback");
    swapRequests[orderHash].status = SwapRequestStatus.PENDING;
    swapRequests[orderHash].blocknumber = block.number;
    emit RollbackSwapRequest(msg.sender, orderHash);
}
```

Listing 2.30: src/Swap.sol

6. The swap process requires the maker to first call makerConfirmSwapRequest() and complete the transfer. At this point, the taker must call confirmSwapRequest() or

cancelSwapRequest() within a certain timeframe. Otherwise, the transaction status will remain stuck in the state MAKER_CONFIRMED, potentially causing economic losses for the makers.

7. In Swap, AssetFeeManager, AssetIssuer, and AssetRebalancer, tokensets calculations use 10**8 as a fixed division factor. To prevent calculation errors, participants must ensure that the decimals for inAmount or outAmount are set to 8. Non-compliance may lead to incorrect results.
8. The orderSigner in the contract USSI must be an EOA address, as signatures require confirmation by the orderSigner. If it is a contract address, the contract USSI will call the function orderSigner.isValidSignature(). Although the orderSigner is set by the owner, its safety cannot be confirmed during this audit if it is a contract, as the contract is out of scope. This could lead to unexpected errors caused by the orderSigner.
9. The protocol includes three types of transaction hashes: Swap.inTxHashs, Swap.outTxHashs, and USSI.redemitTxHashs. These transaction hashes serve as alternatives for token transfers within the contract. The validation of these hashes is performed off-chain. The receiver can verify the transaction using transfer amount, receiver address, and order hash. Incorrect validation may lead to token loss.
10. During rebalance, a swap request is initiated. Token transfers occur off-chain, with transfer details recorded in the contract Swap. Once the swap request is confirmed, the owner verifies the asset transfer, and then rebalancing is performed based on the order Info.

Feedback from the project All the privileged accounts are goverened by MPC custodial wallets to ensure safety.

2.3.2 Withdrawal may not occur within the expected timeframe

Introduced by Version 1

Description In the contract AssetLocking, calling unlock() followed by withdraw() after the cooldown period allows users to withdraw their funds. However, if the previously unlocked funds are not withdrawn, invoking unlock() again resets the cooldown for those funds.

```solidity
function withdraw(address token, uint256 amount) external whenNotPaused {
    LockData storage lockData = lockDatas[token][msg.sender];
    require(lockData.cooldownAmount > 0, "nothing to withdraw");
    require(lockData.cooldownEndTimestamp <= block.timestamp, "coolingdown");
    require(lockData.amount <= lockData.cooldownAmount, "no enough balance to withdraw");
    IERC20(token).safeTransfer(msg.sender, amount);
    lockData.cooldownAmount -= amount;
    LockConfig storage lockConfig = lockConfigs[token];
    lockConfig.totalCooldown -= amount;
    emit Withdraw(msg.sender, token, amount);
}
```

Listing 2.31: src/AssetLocking.sol

Feedback from the project This is by design.

2.3.3 Limited support tokens in the protocol

Introduced by Version 1

Description Currently, there is no whitelist for tokens used in the protocol. When using unsupported weird tokens, such as tokens withcallbacks (like ERC-777, or ERC-721 NFTs misused as ERC-20 tokens), transfer-on-fee tokens, elastic supply tokens, and rebasing tokens, the protocol may not function properly and may potentially subject to attacks. Additionally, centralized tokens like USDT and USDC, which have a function pause(), could indirectly cause a DoS on the protocol if paused.If a user is blacklisted by a token like USDT, they will not be able to withdraw USDT or any other tokens, potentially resulting in economic losses. In summary, the protocol maintainers should choose the tokens to be supported properly for the trusted roles.

Feedback from the project We have added token whitelists in Verison 2 to limit the supported tokens in SSI Protocol.

2.3.4 Inconsistency of participant permissions in contracts AssetIssuer and USSI

Introduced by Version 1

Description AssetIssuer's PARTICIPANT_ROLE and USSI's PARTICIPANT_ROLE are distinct. Possession of a participation role in USSI without the corresponding role in AssetIssuer prevents minting in AssetIssuer, and the reverse applies.

```
function addMintRequest(uint256 assetID, OrderInfo memory orderInfo) external whenNotPaused
    returns (uint) {
   require(_participants[assetID].contains(msg.sender), "msg sender not a participant");
```

Listing 2.32: src/AssetIssuer.sol

```
function applyMint(HedgeOrder calldata hedgeOrder, bytes calldata orderSignature) external
    onlyRole(PARTICIPANT_ROLE) whenNotPaused {
   require(hedgeOrder.requester == msg.sender, "msg sender is not requester");
   bytes32 orderHash = keccak256(abi.encode(hedgeOrder));
   checkHedgeOrder(hedgeOrder, orderHash, orderSignature);
   require(hedgeOrder.orderType == HedgeOrderType.MINT, "order type not match");
```

Listing 2.33: src/USSI.sol

Feedback from the project This is by design.

2.3.5 Additional checks for rescuing funds

Introduced by Version 1

Description The function withdraw() is designed to retrieve the entire balance of any token from the contract, primarily to rescue funds that are stuck. The design requires that none of the tokens in the assetTokens array are in the issuing state. Otherwise, the function will revert.

```
function withdraw(address[] memory tokenAddresses) external onlyOwner {
    IAssetFactory factory = IAssetFactory(factoryAddress);
    uint256[] memory assetIDs = factory.getAssetIDs();
    for (uint i = 0; i < assetIDs.length; i++) {
        IAssetToken assetToken = IAssetToken(factory.assetTokens(assetIDs[i]));
        require(!assetToken.issuing(), "is issuing");
    }
    for (uint i = 0; i < tokenAddresses.length; i++) {
        if (tokenAddresses[i] != address(0)) {
            IERC20 token = IERC20(tokenAddresses[i]);
            token.safeTransfer(owner(), token.balanceOf(address(this)));
        }
    }
}
```

Listing 2.34: src/AssetIssuer.sol

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS