Zellic

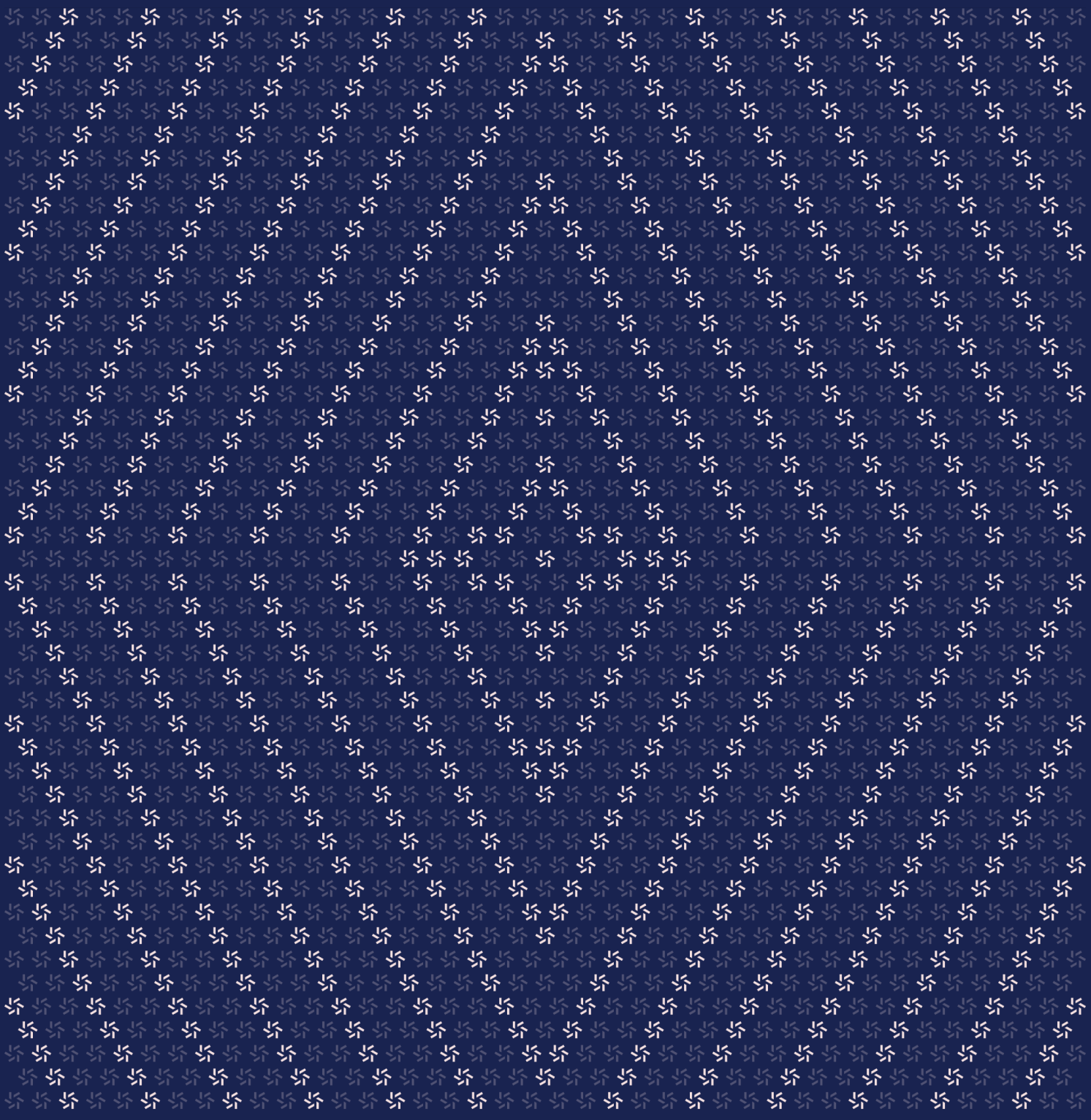
Prepared for
Jivvva
SoSoValue

Prepared by
Jinseo Kim
Sylvain Pelissier
Zellic

February 5, 2025

SSI Protocol

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for SoSoValue from January 20th to January 27th, 2025. During this engagement, Zellic reviewed SSI Protocol's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are all access-control modifiers implemented correctly and working as expected?
- Are all upgradability patterns implemented correctly and working as expected?
- Do changing issuer and rebalancer work as expected?
- Do issuing, rebalancing, and fee-burning features work as expected?
- Is the token set and basket of the asset token updated correctly?
- Do staking/locking contracts correctly implement the cooldown mechanism?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Multichain token transaction validation
- Order-signature security
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped SSI Protocol contracts, we discovered seven findings. No critical issues were found. Three findings were of medium impact and four were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of

SoSoValue in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	3
Low	4
Informational	0

2. Introduction

2.1. About SSI Protocol

SoSoValue contributed the following description of SSI Protocol:

> 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.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect

its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

SSI Protocol Contracts

Type	Solidity
Platform	EVM-compatible
Target	ssi-protocol
Repository	https://github.com/SoSoValueLabs/ssi-protocol ↗
Version	`fe3504a11063a56a337535d2ea00493269d85ac7`

Programs	`AssetController`
	`AssetIssuer`
	`AssetFactory`
	`AssetFeeManager`
	`AssetLocking`
	`AssetRebalancer`
	`AssetToken`
	`StakeFactory`
	`StakeToken`
	`Swap`
	`USSI`
	`Utils`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of one and a half person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Jinseo Kim
Engineer
jinseo@zellic.io ↗

Sylvain Pelissier
Engineer
sylvain@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 20, 2025	Start of primary review period
January 27, 2025	End of primary review period

3. Detailed Findings

3.1. Missing duplicate-token check

Target	AssetFeeManager		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The function `containTokenset` works by looping over a set of tokens (the parameter `b`) to ensure a larger token set (the parameter `a`) contains all the tokens in `b` and contains more than the amount in `b`:

```
function containTokenset(Token[] memory a, Token[] memory b)
    internal pure returns (bool) {
    uint k;
    for (uint i = 0; i < b.length; i++) {
        k = a.length;
        for (uint j = 0; j < a.length; j++) {
            if (isSameToken(b[i], a[j])) {
                if (a[j].amount < b[i].amount) {
                    return false;
                }
                k = j;
                break;
            }
        }
        if (k == a.length) {
            return false;
        }
    }
    return true;
}
```

Therefore, the given small token set must not have any duplication, or the returned result may be incorrect. For example, if the large token set contains a token with an amount of 10, and the small token set contains two identical tokens, each with an amount of 6, the `containTokenset` function will return true even though the large set does not contain the amount of the smaller set.

Impact

The function `containTokenset` does not perform a duplicate check on the given token set. This function is called by `addRebalanceRequest`, which computes a new token basket set and finally calls `hasDuplicates` to verify if it contains any duplicate tokens. However, the function `addBurnFeeRequest` receives an order and computes the fees associated to the `sellTokenset`. Then, the function checks if the asset has enough tokens to pay the fees by calling the `containTokenset` function, but it does not check for any duplicates. The resulting computed fees may then be larger than the asset fees.

Recommendations

We recommend implementing a correct duplicate check that could be integrated to `containTokenset`.

Remediation

This issue has been acknowledged by SoSoValue, and a fix was implemented in commit `7ddd9c5d` ↗.

3.2. Incorrect length calculation on `subTokenset`

Target	Utils		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The function `subTokenset` implements the subtract operation of token sets:

```
function subTokenset(Token[] memory a_, Token[] memory b)
    internal pure returns (Token[] memory) {
    Token[] memory a = copyTokenset(a_);
    uint newLength = a.length;
    uint k;
    for (uint i = 0; i < b.length; i++) {
        k = a.length;
        for (uint j = 0; j < a.length; j++) {
            if (isSameToken(b[i], a[j])) {
                require(a[j].amount >= b[i].amount, "a.amount less than
    b.amount");
                a[j].amount -= b[i].amount;
                if (a[j].amount == 0) {
                    newLength -= 1;
                }
                k = j;
                break;
            }
        }
        require(k < a.length, "a not contains b");
    }
    Token[] memory res = new Token[](newLength);
    k = 0;
    for (uint i = 0; i < a.length; i++) {
        if (a[i].amount > 0) {
            res[k++] = a[i];
        }
    }
    return res;
}
```

This function subtracts the token set `b` from the token set `a` and returns the result-token set. Internally, it modifies the given token set `a` with in-place manner, creates a new result-token set array, and then copies the nonzero tokens to the new array.

To calculate the length of the result-token–set array, the function uses the variable `newLength`. The initial value of this variable is the length of the array `a`, and this variable decrements when the function finds that subtracting the token leads the element in `a` to have a zero amount.

However, this logic is not perfect for calculating the length of a result array, because

- one might include a token with a zero balance in the token set `a`. While this influences the initial value of `newLength`, this will not be copied to the result-token set, resulting in an uninitialized token in the array.
- one might also include a token with a zero balance in the token set `a` and then duplicates the same zero-balance token in the token set `b` multiple times. This would reduce the `newLength` more than the number of the tokens with zero balance, having some tokens omitted in the result-token set.

Impact

The `subTokenset` function may return the token set with uninitialized tokens or without some tokens.

Recommendations

Consider improving the way `newLength` is handled or manually calculating the length of the result-token set after the subtraction.

Remediation

This issue has been acknowledged by SoSoValue, and a fix was implemented in commit `7ddd9c5d` ↗.

3.3. Incomplete duplicate-token check

Target	Utils		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In the contracts, a token is represented by the following structure:

```
struct Token {
    string chain;
    string symbol;
    string addr;
    uint8 decimals;
    uint256 amount;
}
```

To convert the `addr` field to the type address, the function `stringToAddress` is used at many places. It converts the hexadecimal string to a number of type `address`:

```
function stringToAddress(string memory str) internal pure returns (address) {
    bytes memory strBytes = bytes(str);
    require(strBytes.length == 42, "Invalid address length");
    bytes memory addrBytes = new bytes(20);

    for (uint i = 0; i < 20; i++) {
        addrBytes[i] = bytes1(hexCharToByte(strBytes[2 + i * 2])
    * 16 + hexCharToByte(strBytes[3 + i * 2]));
    }

    return address(uint160(bytes20(addrBytes)));
}
```

However, the function does not check that the two first characters of the string are `"0x"`. Thus, those characters can have any values, but the address returned by `stringToAddress` will still be the same. This allows bypassing the check of `hasDuplicates`,

```
function hasDuplicates(Token[] memory a) internal pure returns (bool) {
    uint256[] memory b = new uint256[](a.length);
```

```
    for (uint i = 0; i < a.length; i++) {
        b[i] = uint256(calcTokenHash(a[i]));
    }
    uint256[] memory c = Arrays.sort(b);
    for (uint i = 0; i < c.length - 1; i++) {
        if (c[i] == c[i+1]) {
            return true;
        }
    }
    return false;
}
```

since the `calcTokenHash` function uses the full string in the hash calculation. Using duplicate tokens and just changing those two bytes allows having duplicate tokens in the basket.

This problem is also in the hexadecimal address string itself, since uppercase and lowercase letters would result in the same address value but a different hash.

Impact

The duplicate check is used in the AssetRebalancer contract by the `addRebalanceRequest` to ensure the token set has no duplicate token. Bypassing this check with the previous behavior would allow to incorrectly update the token's amount, as seen in Finding 3.1. ↗.

The `calcTokenHash` computation also breaks the whitelist checks. A token with a different uppercase or lowercase letter in its address string than an identical whitelisted token will not be accepted by the `checkOrderInfo` function even if it should.

Recommendations

The `addr` attribute should be of type `address` to simplify the comparison. It should be converted to hexadecimal string when needed to be displayed.

Remediation

This issue has been acknowledged by SoSoValue, and a fix was implemented in commit `9f023219` ↗.

3.4. Incomplete chain comparison

Target	Swap, AssetIssuer, AssetFeeManager		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

During the check of a token set by the function `checkTokenset`, the `chain` string from the `Token` structure is compared with the `chain` from the swap:

```
function checkTokenset(Token[] memory tokenset, string[] memory addressList)
    internal view {
    require(tokenset.length == addressList.length, "tokenset length not match
    addressList length");
    for (uint i = 0; i < tokenset.length; i++) {
        require(bytes32(bytes(tokenset[i].chain)) == bytes32(bytes(chain)),
    "chain not match");
        address tokenAddress = Utils.stringToAddress(tokenset[i].addr);
        require(tokenAddress != address(0), "zero token address");
        address receiveAddress = Utils.stringToAddress(addressList[i]);
        require(receiveAddress != address(0), "zero receive address");
    }
}
```

The casting of `bytes32(bytes(chain))` truncates the string to exactly 32 bytes, and the comparison is done only on the first 32 bytes of the `chain` string, further bytes will be skipped. This check is also implemented in the `addMintRequest`, `rejectMintRequest`, `confirmMintRequest`, `addRedeemRequest`, and `addBurnFeeRequest` functions.

Impact

In the case that the chains have more than 32 bytes, some tokens with different chains, but a common 32-byte prefix, would be considered equal by the check, allowing them in the set even though they should not.

Recommendations

It would be less error-prone to declare the `chain` as a `bytes32` instead of a string. It would also save gas by removing the comparisons, which are currently made with the Keccak hash function, like in

the `checkHedgeOrder` function of the USSI contract:

```
function checkHedgeOrder(HedgeOrder calldata hedgeOrder, bytes32 orderHash,
    bytes calldata orderSignature) public view {
    require(keccak256(abi.encode(chain)) ==
    keccak256(abi.encode(hedgeOrder.chain)), "chain not match");
    //...
```

Remediation

This issue has been acknowledged by SoSoValue, and a fix was implemented in commit 2cc062d0 ↗.

3.5. Incorrect token comparison

Target	Utils		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Two tokens are compared with the `isSameToken` function, which calls the `calcTokenHash` function on the `Token` structure:

```
function calcTokenHash(Token memory token) internal pure returns (bytes32) {
    return keccak256(abi.encodePacked(token.chain, token.symbol, token.addr,
    token.decimals));
}
```

However, the `Token` structure is encoded with the function `abi.encodePacked`. As seen in Finding 3.3. ↗, the structure contains dynamic typed fields. Thus, the encoding is not unique and allows collisions in the hash function. Here is a test demonstrating the issue:

```
function test_HashCollision() public {
    Token memory token1 = Token({
        chain: "BTC",
        symbol: "2200",
        addr: vm.toString(address(WBTC)),
        decimals: WBTC.decimals(),
        amount: 0
    });

    Token memory token2 = Token({
        chain: "BTC2",
        symbol: "200",
        addr: vm.toString(address(WBTC)),
        decimals: WBTC.decimals(),
        amount: 0
    });

    assertTrue(Utils.isSameToken(token1, token2));
}
```

Impact

The issue seems difficult to exploit since the address often converts with the `stringToAddress` function, which enforces the address length. Nevertheless, this could be a source of errors and confusion.

Recommendations

The internal function `abi.encode` should be used instead of `abi.encodePacked`. More generally, depending on the external usage, only the address could be compared since two tokens that share the same address can be considered as identical.

Remediation

This issue has been acknowledged by SoSoValue.

3.6. The `getOrderHashs` function returns the array growing indefinitely

Target	USSI		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The following `getOrderHashs` function returns the list of all order hashes stored in the contract:

```
function getOrderHashs() external view returns (bytes32[] memory orderHashs_)
    {
    orderHashs_ = new bytes32[](orderHashs.length());
    for (uint i = 0; i < orderHashs.length(); i++) {
        orderHashs_[i] = orderHashs.at(i);
    }
}
```

However, the list of order hashes grows indefinitely, posing a possibility that this function reverts after the number of order hashes exceeds some threshold determined by the gas.

Impact

The `getOrderHashs` function may revert in the future, assuming the list of order hashes constantly grows. Although this function is not used by other contracts in the scope of this audit and therefore this won't affect the on-chain business logic, off-chain infrastructure depending on this function may be affected by this finding.

Recommendations

Consider removing the function if possible.

Remediation

This issue has been acknowledged by SoSoValue.

3.7. Lack of domain separation allows signature replay

Target	Swap, USSI		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The `checkOrderInfo` of a Swap contract is used to check the validity of an order by verifying it was signed by its maker address. To prevent replay, the order hash is stored in the Swap contract, and an order is not accepted if its hash has been already validated. The signature is verified with the function `SignatureChecker.isValidSignatureNow` from the OpenZepplin library:

```
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;
    }
    // ...
```

This function checks if a hash was properly signed by the given address. However, the hash does not include any domain separation. Thus, the same signature can be replayed to other Swap contracts and the order will be seen as valid.

Impact

The AssetFactory contract stores the swap associated to each asset token. If two different asset tokens have different Swap contracts, then an order signature can be replayed to the `addMintRequest` function and will be seen as valid. The Swap can also be changed with the

`setSwap` function. For a new Swap, all previous orders could be replayed as well. The two swap requests for two different assets will be placed even if a single order was validated.

Another less critical problem is also a lack of domain separation between Swap and USSI signatures. Swap verifies the `Order` structure and the USSI verifies the `HedgeOrder` struct. Fortunately, the two structures are different and one signature cannot be replayed to another contract. However, it is a good practice to have domain separation, to avoid reusing one signature from a contract to another.

Recommendations

The hash should include the Swap contract address to prevent this replay, like in EIP-712 ↗.

Remediation

This issue has been acknowledged by SoSoValue.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Test suite

The existing testing suite used during the assessment is limited in its coverage and does not fully test all components of the smart contracts. This limitation prevented comprehensive testing and identification of potential issues, particularly in terms of negative testing scenarios. Some of the findings reported in this assessment like Findings 3.1. ↗, 3.3. ↗, and 3.5. ↗ could have been detected if the testing suite had been more comprehensive. To ensure the robustness of the smart contracts, we recommend that SoSoValue expands the testing suite to cover all functionality, including negative testing outlined in the threat model. Especially the Utils contract, which is used in many different places in the code and ensures many invariants, should be intensively tested and fuzzed.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality most likely was not broken by your change to the code.

Remediation

This issue has been acknowledged by SoSoValue, and additional tests were implemented in commit `1107e46b` ↗.

4.2. Custom errors could save gas

Solidity allows using custom errors. Using custom errors instead of `revert` or `require` saves gas since the error message used by the revert function is stored on chain. Here is an example usage:

```
error ReblancingError();
error ZeroSupplyError();
function rebalance(Token[] memory inBasket, Token[] memory outBasket)
    external onlyRole(REBALANCER_ROLE) {
    require(rebalancing, "lock rebalance first");
   if (!rebalancing) { revert ReblancingError();}
    require(totalSupply() > 0, "zero supply");
   if (totalSupply() == 0) { revert ZeroSupplyError();}
    Token[] memory newBasket = Utils.addTokenset(Utils.subTokenset(basket_,
    outBasket), inBasket);
    Token[] memory newTokenset = Utils.muldivTokenset(newBasket,
    10**decimals(), totalSupply());
    setBasket(newBasket);
    setTokenset(newTokenset);
}
```

4.3. Redundant checks

The contract implements some unnecessary checks. For example, before the calls to the `safeTransferFrom` function, the following checks are implemented at seven different places:

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

However, the OpenZeppelin implementation of `safeTransferFrom` also implements this check internally.

Each of those checks consume gas since the revert string has to be stored on chain.

There are also redundant checks regarding the order validation. The `addSwapRequest` function validates the order by calling `checkOrderInfo`:

```
function addSwapRequest(OrderInfo memory orderInfo, bool inByContract,
    bool outByContract) external onlyRole(TAKER_ROLE) whenNotPaused {
        uint code = checkOrderInfo(orderInfo);
        require(code == 0, "order not valid");
```

However, this function is called by the functions `addRedeemRequest`, `addRebalanceRequest`, `addMintRequest`, and `addBurnFeeRequest`, which all also call `checkOrderInfo` before calling `addSwapRequest`.

Removing those checks but still ensuring that the contract behaves as intended by comprehensive testing would save some gas.

4.4. No storage gap for upgradable contracts

When we observe contracts inherited by upgradable contracts that are not deployed to the chain, we typically recommend to implement storage gaps ↗ so it is easy to add additional storage variables to the contract while not corrupting the storage layout of other contracts.

In this codebase, the AssetController contract falls to this case; it is inherited by AssetFeeManager, AssetIssuer, and AssetRebalancer. However, they do not implement the storage gap, and these contracts are already deployed on chain.

Upgrading these contracts by modifying the code of AssetController should be done cautiously because adding a storage variable to AssetController may corrupt the storage layouts of child contracts.

4.5. Centralization risk

In the Swap contract, the orders are signed by an address with the `MAKER_ROLE`. If one of those addresses is compromised, it would also allow signing any orders, allowing minting, burning fees, and redeeming operations until the role is revoked by the owner of the Swap contract. In addition, the owner can revoke a maker before a participant uses their orders and thus grief any participants.

The above introduces a centralization risk that users should be aware of, as it grants a single point of control over the system.

We recommend that this centralization risk be clearly documented for users so that they are aware of the extent of the owner's control over the contract. This can help users make informed decisions about their participation in the project. Additionally, clear communication about the circumstances in which the owner may exercise these powers can help build trust and transparency with users. Therefore, it is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for owner access.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: AssetFactory.sol

Function: `createAssetToken(Asset asset, uint256 maxFee, address issuer, address rebalancer, address feeManager, address swap_)`

This function creates an asset-token contract.

Inputs

- `asset`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must have an asset ID that does not exist.
  - **Impact**: Asset to be created.
- `maxFee`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be less than `10 ** feeDecimals`.
  - **Impact**: Maximum fee of the asset token.
- `issuer`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: Issuer of the asset token.
- `rebalancer`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: Rebalancer of the asset token.
- `feeManager`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: Fee manager of the asset token.
- `swap_`
  - **Control**: Fully controlled by the caller.

- **Constraints**: None.
- **Impact**: Swap contract of the asset token.

Branches and code coverage

Intended branches

- Create the asset-token contract.
  - ☑ Test coverage
- Grant the role to the issuer, rebalancer, and fee manager.
  - ☑ Test coverage
- Store the information about the asset token.
  - ☑ Test coverage

Negative behavior

- The caller is the owner.
  - ☐ Negative test
- Asset ID does not exist.
  - ☐ Negative test
- Maximum fee is less than `10 ** feeDecimals`.
  - ☐ Negative test

Function: `setFeeManager(uint256 assetID, address feeManager)`

This function sets the fee manager of the asset.

Inputs

- `assetID`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be an asset ID.
  - **Impact**: Specifies the asset to change the fee manager.
- `feeManager`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: New fee manager of the given asset.

Branches and code coverage

Intended branches

- Revoke the fee manager role of the old fee manager.
  - ☐ Test coverage
- Grant the fee manager role to the new fee manager.
  - ☐ Test coverage

Negative behavior

- The caller is the owner.
  - ☐ Negative test
- Given asset ID is valid.
  - ☐ Negative test
- Asset token does not have a pending fee-burning process.
  - ☐ Negative test

Function: `setIssuer(uint256 assetID, address issuer)`

This function sets the issuer of the asset.

Inputs

- `assetID`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be an asset ID.
  - **Impact**: Specifies the asset to change the issuer.
- `issuer`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: New issuer of the given asset.

Branches and code coverage

Intended branches

- Revoke the issuer role of the old issuer.
  - ☐ Test coverage

- Grant the issuer role to the new issuer.
  - ☐ Test coverage

Negative behavior

- The caller is the owner.
  - ☐ Negative test
- Given asset ID is valid.
  - ☐ Negative test
- Asset token does not have a pending issuing process.
  - ☐ Negative test

Function: `setRebalancer(uint256 assetID, address rebalancer)`

This function sets the rebalancer of the asset.

Inputs

- `assetID`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be an asset ID.
  - **Impact**: Specifies the asset to change the rebalancer.
- `rebalancer`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: New rebalancer of the given asset.

Branches and code coverage

Intended branches

- Revoke the rebalancer role of the old rebalancer.
  - ☐ Test coverage
- Grant the rebalancer role to the new rebalancer.
  - ☐ Test coverage

Negative behavior

- The caller is the owner.

- ☐ Negative test

- Given asset ID is valid.
  - ☐ Negative test
- Asset token does not have a pending rebalancing process.
  - ☐ Negative test

5.2. Module: AssetFeeManager.sol

Function: `addBurnFeeRequest(uint256 assetID, OrderInfo orderInfo)`

The function is used to add a burn-fee request for a given asset.

Inputs

- `assetID`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The contract is checked to be a fee manager of the asset.
  - **Impact**: A non-manager cannot create a burn request.
- `orderInfo`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The order should not have been used before for the asset swap. The order hash should match the hash computed over the order structure. The signature of the order is verified to have been made by the order maker.
  - **Impact**: Prevents invalid order from being passed or replayed to the same swap.

Branches and code coverage

Intended branches

- A burn request is successfully created for a valid order and a fee manager.
  - ☑ Test coverage

Negative behavior

- A burn request should revert when the contract is not a fee manager for the asset.
  - ☐ Negative test
- A burn request should revert when the order is invalid.
  - ☐ Negative test

- A burn request should revert when the order was already used for the same swap.
  - ☐ Negative test
- A burn request should revert when there are not enough fee tokens.
  - ☐ Negative test
- A burn request should revert when the chains do not match.
  - ☐ Negative test
- A burn request should revert when the out-token list is not the vault.
  - ☐ Negative test
- A burn request should revert when some tokens are not whitelisted.
  - ☐ Negative test

5.3. Module: AssetIssuer.sol

Function: `addMintRequest(uint256 assetID, OrderInfo memory orderInfo, uint256 maxIssueFee)`

The function creates a mint request for the given asset and the given order. The in-token amount of the participant is transferred to the contract.

Inputs

- `assetID`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The sender is checked to be a participant of the asset.
  - **Impact**: A non-participant cannot create a mint request.
- `orderInfo`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The order should not have been used before for the swap. The order hash should match the hash computed over the order structure. The signature of the order is verified to have been made by the order maker.
  - **Impact**: Prevents invalid order from being passed or replayed to the same swap.
- `maxIssueFee`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The max fee is checked to be greater than the issue fee.
  - **Impact**: Prevents using an invalid fee.

Branches and code coverage (including function calls)

Intended branches

- A mint request is successfully created for a valid order and maximum fee.
  - ☑ Test coverage

Negative behavior

- A mint request should revert when the sender is not a participant.
  - ☐ Negative test
- A mint request should revert when the order is invalid.
  - ☐ Negative test
- A mint request should revert when the order was already used for the same swap.
  - ☐ Negative test
- A mint request should revert when the token sets mismatch.
  - ☐ Negative test
- A mint request should revert when the chains in the order, in the factory, or in the token sets do not match.
  - ☐ Negative test
- A mint request should revert when the balance of the in-token set is too low.
  - ☐ Negative test
- A mint request should revert when some tokens are not whitelisted.
  - ☐ Negative test

Function: `addRedeemRequest(uint256 assetID, OrderInfo orderInfo, uint256 maxIssueFee)`

The function creates a redeem request for the given asset and the given order. The in-token set of the order is checked to match the token set of the asset. If successful, the asset-token amount of the participant is transferred to the contract.

Inputs

- `assetID`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The sender is checked to be a participant of the asset.
  - **Impact**: A non-participant cannot create a redeem request.
- `orderInfo`

  - **Control**: Fully controlled by the caller.
  - **Constraints**: The order should not have been used before for the asset swap. The order hash should match the hash computed over the order structure. The signature of the order is verified to have been made by the order maker.
  - **Impact**: Prevents invalid order from being passed or replayed to the same swap.
- `maxIssueFee`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The max fee is checked to be greater than the issue fee.
  - **Impact**: Prevents using an invalid fee.

Branches and code coverage

Intended branches

- A redeem request is successfully created for a valid order and maximum fee. The token balances are updated correctly.
  - ☑ Test coverage

Negative behavior

- A redeem request should revert when the sender is not a participant.
  - ☐ Negative test
- A redeem request should revert when the order is invalid.
  - ☐ Negative test
- A redeem request should revert when the order was already used for the same swap.
  - ☐ Negative test
- A redeem request should revert when the token sets mismatch.
  - ☐ Negative test
- A redeem request should revert when the chains in the order, in the factory, or in the token sets do not match.
  - ☐ Negative test
- A redeem request should revert when the balance of the in-token set is too low.
  - ☐ Negative test
- A redeem request should revert when some tokens are not whitelisted.
  - ☐ Negative test

5.4. Module: AssetLocking.sol

Function: `lock(address token, uint256 amount)`

This function locks a token.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be registered into this contract and under the active epoch.
  - **Impact**: Specifies the token to be locked.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Specifies the amount of tokens to be locked.

Branches and code coverage

Intended branches

- Increase the locked amount of the user.
  - ☑ Test coverage
- Transfer the tokens from the user.
  - ☑ Test coverage

Negative behavior

- Contract is not paused.
  - ☐ Negative test
- The given token is registered to support.
  - ☑ Negative test
- The given token has an active epoch.
  - ☐ Negative test

Function: `unlock(address token, uint256 amount)`

This function unlocks a token.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Specifies the token to be unlocked.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be less than the locked amount of the user.
  - **Impact**: Specifies the amount of tokens to be unlocked.

Branches and code coverage

Intended branches

- Decrease the locked amount of the user.
  - ☑ Test coverage
- Increase the cooldown amount of the user.
  - ☑ Test coverage
- Update the cooldown period of the user.
  - ☑ Test coverage

Negative behavior

- Contract is not paused.
  - ☐ Negative test
- Caller has enough locked amount to unlock.
  - ☑ Negative test

Function: `withdraw(address token, uint256 amount)`

This function withdraws a token after the cooldown period elapsed.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Specifies the token to be withdrawn.

- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be less than the cooldown amount of the caller.
    - **Impact**: Specifies the amount of tokens to be withdrawn.

Branches and code coverage

Intended branches

- Decrease the cooldown amount of the caller.
    - ☑ Test coverage
- Transfer the tokens to the caller.
    - ☐ Test coverage

Negative behavior

- Contract is not paused.
    - ☐ Negative test
- The cooldown period has passed.
    - ☑ Negative test
- The caller has enough of a cooldown amount to withdraw.
    - ☐ Negative test

5.5. Module: AssetRebalancer.sol

Function: `addRebalanceRequest(uint256 assetID, Token[] basket, OrderInfo orderInfo)`

Add the rebalance request to the contract.

Inputs

- `assetID`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be a valid asset ID.
    - **Impact**: Specifies the asset to be rebalanced.
- `basket`
    - **Control**: Fully controlled by the caller.

- **Constraints**: Must be equal to the basket of the asset token.
- **Impact**: Confirms the current basket of the asset token.

- `orderInfo`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be valid `OrderInfo`.
  - **Impact**: Prevents invalid order from being passed or replayed to the same swap.

Branches and code coverage

Intended branches

- Calculate the new basket and token set correctly.
  - ☑ Test coverage
- Create a new swap request handling the order.
  - ☑ Test coverage
- Lock the rebalance feature.
  - ☑ Test coverage

Negative behavior

- The caller is the owner.
  - ☐ Negative test
- Asset token does not rebalance or issue status.
  - ☐ Negative test
- Fee has been collected from the asset token.
  - ☐ Negative test
- Given order is valid.
  - ☐ Negative test

5.6. Module: AssetToken.sol

Function: `burnFeeTokenset(Token[] feeTokenset)`

This function burns the tokens in the fee-token set.

Inputs

- `feeTokenset`

- **Control**: Fully controlled by the caller of AssetFeeManager.
- **Constraints**: Must be contained by the fee-token set.
- **Impact**: Specifies the amount of underlying assets to burn from the fee-token set.

Branches and code coverage

Intended branches

- Calculate the new fee-token set and apply.
  - ☑ Test coverage

Negative behavior

- The caller is a fee manager.
  - ☑ Negative test
- Given set is contained by the fee-token set.
  - ☐ Negative test

Function: `burn(uint256 amount)`

This function burns the asset token.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller of AssetIssuer.
  - **Constraints**: Must be less than the balance of the caller.
  - **Impact**: Specifies the amount of asset tokens to burn.

Branches and code coverage

Intended branches

- Burn the asset token.
  - ☑ Test coverage
- Change the basket.
  - ☑ Test coverage

Negative behavior

- The caller is an issuer.
  - ☐ Negative test

Function: `collectFeeTokenset()`

This function collects a certain fraction of the basket into the fee-token set.

Branches and code coverage

Intended branches

- Calculate the amount of underlying asset to transfer to the fee-token set.
  - ☑ Test coverage
- Transfer the underlying asset from the basket to the fee-token set.
  - ☑ Test coverage
- Recalculate the new token set and apply.
  - ☑ Test coverage
- Update the timestamp when fee is collected.
  - ☑ Test coverage

Negative behavior

- The caller is a fee manager.
  - ☐ Negative test
- Rebalancing and issuing features are not locked.
  - ☐ Negative test

Function: `initTokenset(Token[] tokenset)`

This function initializes the token set of the contract.

Inputs

- `tokenset`
  - **Control**: Fully controlled by the owner of AssetFactory.
  - **Constraints**: None.
  - **Impact**: Token set of the contract.

Branches and code coverage

Intended branches

- Change the token set of the contract.
  - ☑ Test coverage

Negative behavior

- The caller is an admin.
  - ☐ Negative test

Function: `lockBurnFee()`

This function locks the fee-burning feature.

Branches and code coverage

Intended branches

- Set the fee-burning flag to true.
  - ☑ Test coverage

Negative behavior

- The caller is a fee manager.
  - ☐ Negative test
- Fee-burning feature is not locked.
  - ☐ Negative test

Function: `lockIssue()`

This function locks the issuing feature.

Branches and code coverage

Intended branches

- Increment the count of the issuing process.
  - ☑ Test coverage

Negative behavior

- The caller is an issuer.
  - ☐ Negative test
- Contract is not rebalancing.
  - ☐ Negative test

Function: `lockRebalance()`

This function locks the rebalancing feature.

Branches and code coverage

Intended branches

- Set the rebalancing flag to true.
  - ☑ Test coverage

Negative behavior

- The caller is a rebalancer.
  - ☐ Negative test
- Contract is not issuing.
  - ☐ Negative test
- Contract is not rebalancing.
  - ☐ Negative test

Function: `mint(address account, uint256 amount)`

This function mints the asset token.

Inputs

- `account`
  - **Control**: Fully controlled by the caller of AssetIssuer.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: Specifies the account to receive the minted token.
- `amount`
  - **Control**: Fully controlled by the caller of AssetIssuer.
  - **Constraints**: None.

- **Impact**: Specifies the amount of asset token to mint.

Branches and code coverage

Intended branches

- Mint the asset token.
  - ☑ Test coverage
- Change the basket.
  - ☑ Test coverage

Negative behavior

- The caller is an issuer.
  - ☐ Negative test
- Contract is issuing.
  - ☐ Negative test

Function: `rebalance(Token[] inBasket, Token[] outBasket)`

This function rebalances the basket of the asset token.

Inputs

- `inBasket`
  - **Control**: Controlled by the caller of AssetRebalancer.
  - **Constraints**: None.
  - **Impact**: Specifies the list of assets to add to the basket.
- `outBasket`
  - **Control**: Controlled by the caller of AssetRebalancer.
  - **Constraints**: Must be contained by the current basket.
  - **Impact**: Specifies the list of assets to subtract from the basket.

Branches and code coverage

Intended branches

- Calculate and set the new basket.
  - ☑ Test coverage

- Calculate and set the new token set.
  - ☑ Test coverage

Negative behavior

- The caller is a rebalancer.
  - ☐ Negative test
- Rebalancing feature is locked.
  - ☐ Negative test

Function: `unlockBurnFee()`

This function unlocks the fee-burning feature.

Branches and code coverage

Intended branches

- Set the rebalancing flag to false.
  - ☑ Test coverage

Negative behavior

- The caller is a fee manager.
  - ☐ Negative test

Function: `unlockIssue()`

The function unlocks the issuing process.

Branches and code coverage

Intended branches

- Decrement the count of the issuing process if nonzero.
  - ☑ Test coverage

Negative behavior

- The caller is an issuer.
  - ☐ Negative test

Function: `unlockRebalance()`

This function unlocks the rebalancing feature.

Branches and code coverage

Intended branches

- Set the rebalancing flag to false.
  - ☑ Test coverage

Negative behavior

- The caller is a rebalancer.
  - ☐ Negative test

5.7. Module: StakeFactory.sol

Function: `createStakeToken(uint256 assetID, uint48 cooldown)`

This function creates a new StakeToken contract.

Inputs

- `assetID`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be an existing asset ID that does not have corresponding StakeToken.
  - **Impact**: Specifies the asset whose stake token would be created.
- `cooldown`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Specifies the cooldown of the stake token to be created.

Branches and code coverage

Intended branches

- Fetch the address of the asset token from the asset ID.
  - ☑ Test coverage
- Create a StakeToken contract.

☑ Test coverage

- Store the information about the created contract.

☑ Test coverage

Negative behavior

- The caller is the owner.

☐ Negative test

- Given asset ID is valid.

☐ Negative test

- Corresponding stake token does not exist.

☐ Negative test

5.8. Module: StakeToken.sol

Function: `stake(uint256 amount)`

This function stakes a token.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Amount that will be staked.

Branches and code coverage

Intended branches

- Transfer the token from the user.

☑ Test coverage

- Mint the token.

☑ Test coverage

Negative behavior

- Contract is not paused.

☑ Negative test

Function: `unstake(uint256 amount)`

This function unstakes the staked token.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be less than the balance of the caller.
  - **Impact**: Amount that will be unstaked.

Branches and code coverage

Intended branches

- Increase the cooldown amount of the caller.
  - ☑ Test coverage
- Update the cooldown timestamp of the caller.
  - ☑ Test coverage
- Burn the tokens of the caller.
  - ☑ Test coverage

Negative behavior

- Contract is not paused.
  - ☐ Negative test
- Amount is nonzero.
  - ☐ Negative test
- Amount is less than the balance of the user.
  - ☐ Negative test

Function: `withdraw(uint256 amount)`

This function withdraws the unstaked tokens.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.

- **Constraints**: Must be less than the amount under the cooldown period.
- **Impact**: Amount of tokens that would be transferred to the caller.

Branches and code coverage

Intended branches

- Decrease the cooldown amount of user.
  - ☑ Test coverage
- Transfer the tokens to the user.
  - ☑ Test coverage

Negative behavior

- Contract is not paused.
  - ☐ Negative test
- Amount is nonzero.
  - ☐ Negative test
- Amount is less than the cooldown amount of the user.
  - ☐ Negative test
- Cooldown period has passed.
  - ☑ Negative test

5.9. Module: Swap.sol

Function: `addSwapRequest(OrderInfo orderInfo, bool inByContract, bool outByContract)`

This function adds a new swap request.

Inputs

- `orderInfo`
  - **Control**: Controlled by the caller of the contracts that use the swap.
  - **Constraints**: Must be a valid order.
  - **Impact**: Specifies the order to process.
- `inByContract`
  - **Control**: Controlled by the caller (contracts).

- **Constraints**: None.
- **Impact**: Specifies whether the contract receives ERC-20 tokens for this swap or not.

- `outByContract`
  - **Control**: Controlled by the caller (contracts).
  - **Constraints**: None.
  - **Impact**: Specifies whether the contract sends ERC-20 tokens for this swap or not.

Branches and code coverage

Intended branches

- Add the order to the storage.
  - ☑ Test coverage

Negative behavior

- The caller is a taker.
  - ☐ Negative test
- Given order is valid.
  - ☐ Negative test
- Given input and output tokens are valid if `inByContract` or `outByContract` is true, respectively.
  - ☐ Negative test

Function: `cancelSwapRequest(OrderInfo orderInfo)`

This function cancels the swap if it is before the maker confirms the request.

Inputs

- `orderInfo`
  - **Control**: Controlled by the caller of the contracts that use the swap.
  - **Constraints**: Must be a pending order.
  - **Impact**: Specifies the order to cancel.

Branches and code coverage

Intended branches

- Change the status of the order to canceled.
  - ☑ Test coverage

Negative behavior

- The caller is a taker.
  - ☐ Negative test
- Given order hash matches to the actual hash.
  - ☐ Negative test
- Given order is requested by the caller.
  - ☐ Negative test
- Maker confirm delay has passed.
  - ☐ Negative test

Function: `checkOrderInfo(OrderInfo orderInfo)`

This function validates the given order.

Inputs

- `orderInfo`
  - **Control**: Controlled by the caller of the contracts that use the swap.
  - **Constraints**: Must be valid.
  - **Impact**: Specifies the order to validate.

Branches and code coverage

Intended branches

- Validate the order.
  - ☐ Test coverage

Negative behavior

- Deadline of the order has not passed.
  - ☐ Negative test

- Order hash in the `OrderInfo` matches the actual hash.
  - ☐ Negative test
- Given order is signed by the maker.
  - ☐ Negative test
- Order is not duplicated.
  - ☐ Negative test
- Maker of the order has the maker role.
  - ☐ Negative test
- Output addresses are whitelisted.
  - ☐ Negative test
- Order is submitted to the correct chain.
  - ☐ Negative test
- Input and output tokens are whitelisted.
  - ☐ Negative test

Function: `confirmSwapRequest(OrderInfo orderInfo, bytes[] inTxHashs)`

This function confirms and finalizes the swap.

Inputs

- `orderInfo`
  - **Control**: Controlled by the caller of the contracts that use the swap.
  - **Constraints**: Must be a valid order.
  - **Impact**: Specifies the order to confirm.
- `inTxHashs`
  - **Control**: Controlled by the caller of the contracts that use the swap.
  - **Constraints**: Must have a number of elements equal to the number of inputs.
  - **Impact**: Specifies the input transactions if `inByContract` is false.

Branches and code coverage

Intended branches

- Change the status of the order to maker-confirmed.
  - ☑ Test coverage

- Execute the input transactions if `inByContract` is true.
  - ☑ Test coverage

Negative behavior

- The caller is a taker.
  - ☐ Negative test
- Given order hash matches the actual hash.
  - ☐ Negative test
- Given order is requested by the caller.
  - ☐ Negative test
- Given order is maker-confirmed.
  - ☐ Negative test

Function: `forceCancelSwapRequest(OrderInfo orderInfo)`

This function forcefully cancels the swap after the expiration of the swap.

Inputs

- `orderInfo`
  - **Control**: Controlled by the caller of contracts that use the swap.
  - **Constraints**: Must be a pending or maker-confirmed order.
  - **Impact**: Specifies the order to cancel.

Branches and code coverage

Intended branches

- Change the status of the order to force-canceled.
  - ☑ Test coverage

Negative behavior

- The caller is a taker.
  - ☐ Negative test
- Given order hash matches the actual hash.
  - ☐ Negative test
- Expiration delay has passed.

- ☐ Negative test

Function: `makerConfirmSwapRequest(OrderInfo orderInfo, bytes[] outTxHashs)`

This function confirms the swap from the maker's side.

Inputs

- `orderInfo`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a pending order.
  - **Impact**: Specifies the order to confirm.
- `outTxHashs`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must have a number of elements equal to the number of outputs.
  - **Impact**: Specifies the output transactions if `outByContract` is false.

Branches and code coverage

Intended branches

- Change the status of the order to maker-confirmed.
  - ☑ Test coverage
- Execute the output transactions if `outByContract` is true.
  - ☑ Test coverage

Negative behavior

- The caller is a maker.
  - ☐ Negative test
- Given order hash matches the actual hash.
  - ☐ Negative test
- The caller is the maker of the given order.
  - ☐ Negative test

Function: `makerRejectSwapRequest(OrderInfo orderInfo)`

This function rejects the swap from the maker's side.

Inputs

- `orderInfo`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a pending order.
  - **Impact**: Specifies the order to cancel.

Branches and code coverage

Intended branches

- Change the status of the order to force-canceled.
  - ☑ Test coverage

Negative behavior

- The caller is a maker.
  - ☐ Negative test
- Given order hash matches the actual hash.
  - ☐ Negative test
- The caller is the maker of the given order.
  - ☐ Negative test

Function: `rollbackSwapRequest(OrderInfo orderInfo)`

This function rolls back the swap after it is maker-confirmed.

Inputs

- `orderInfo`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a maker-confirmed order.
  - **Impact**: Specifies the order to roll back.

Branches and code coverage

Intended branches

- Change the status of the order to pending.
  - ☑ Test coverage

Negative behavior

- The caller is a taker.
  - ☑ Negative test
- Given order hash matches the actual hash.
  - ☐ Negative test
- Given order is requested by the caller.
  - ☐ Negative test
- Given order is maker-confirmed.
  - ☐ Negative test
- `outByContract` is false.
  - ☐ Negative test

5.10. Module: USSI.sol

Function: `applyMint(HedgeOrder hedgeOrder, bytes orderSignature)`

This function adds the minting order.

Inputs

- `hedgeOrder`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid minting hedge order.
  - **Impact**: Specifies the hedge order to add.
- `orderSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid signature for the hash.
  - **Impact**: Specifies the signature for the hedge order.

Branches and code coverage

Intended branches

- Add the order to the storage.
  - ☑ Test coverage
- Receive the asset token from the caller.
  - ☑ Test coverage

Negative behavior

- The caller is a participant.
  - ☑ Negative test
- The caller is the requester of the given hedge order.
  - ☐ Negative test
- Given hedge order is valid.
  - ☐ Negative test
- Given hedge order mints the token.
  - ☐ Negative test
- Rebalancing is not locked on the corresponding asset token.
  - ☐ Negative test
- Fee is recently collected on the corresponding asset token.
  - ☐ Negative test

Function: `applyRedeem(HedgeOrder hedgeOrder, bytes orderSignature)`

This function adds the redemption order.

Inputs

- `hedgeOrder`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid redemption hedge order.
  - **Impact**: Specifies the hedge order to add.
- `orderSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid signature for the hash.
  - **Impact**: Specifies the signature for the hedge order.

Branches and code coverage

Intended branches

- Add the order to storage.
  - ☑ Test coverage
- Receive the USSI token from the caller.
  - ☑ Test coverage

Negative behavior

- The caller is a participant.
  - ☑ Negative test
- The caller is the requester of the given hedge order.
  - ☐ Negative test
- Given hedge order is valid.
  - ☐ Negative test
- Given hedge order redeems the token.
  - ☐ Negative test

Function: `checkHedgeOrder(HedgeOrder hedgeOrder, byte[32] orderHash, bytes orderSignature)`

This function checks if the given hedge order is valid.

Inputs

- `hedgeOrder`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid hedge order.
  - **Impact**: Specifies the hedge order to be validated.
- `orderHash`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Specifies the hash of the hedge order.
- `orderSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid signature for the order hash by the order signer.

- **Impact**: Specifies the signature of the hedge order.

Branches and code coverage

Intended branches

- Validate the given order hash.
  - ☑ Test coverage

Negative behavior

- Given hedge order is submitted to correct chain.
  - ☐ Negative test
- Specified asset is supported if the given order mints a token.
  - ☐ Negative test
- Specified redeem token is equal to the configured token if the given order redeems tokens.
  - ☐ Negative test
- Deadline of order has not passed.
  - ☐ Negative test
- Order is not duplicated.
  - ☐ Negative test
- The order is correctly signed by the order signer.
  - ☐ Negative test

Function: `confirmMint(byte[32] orderHash)`

This function confirms the minting order.

Inputs

- `orderHash`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the order hash of the pending order.
  - **Impact**: Specifies the order to be confirmed.

Branches and code coverage

Intended branches

- Change the status of the order.
  - ☑ Test coverage
- Burn the asset token.
  - ☑ Test coverage
- Mint the USSI token.
  - ☑ Test coverage

Negative behavior

- The caller is the owner.
  - ☐ Negative test
- Given order is the minting order.
  - ☐ Negative test
- Given order is pending.
  - ☐ Negative test

Function: `confirmRedeem(byte[32] orderHash, byte[32] txHash)`

This function confirms the redemption order.

Inputs

- `orderHash`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the order hash of the pending order.
  - **Impact**: Specifies the order to be confirmed.
- `txHash`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Specifies the transaction that processed the redemption.

Branches and code coverage

Intended branches

- Change the status of the order.
  - ☑ Test coverage
- Transfer the redeem token if the transaction hash is not given.
  - ☑ Test coverage
- Burn the redeemed USSI tokens.
  - ☑ Test coverage

Negative behavior

- The caller is the owner.
  - ☐ Negative test
- Given order is the redemption order.
  - ☐ Negative test
- Given order is pending.
  - ☐ Negative test

Function: `rejectMint(byte[32] orderHash)`

This function rejects the minting order.

Inputs

- `orderHash`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the order hash of the pending order.
  - **Impact**: Specifies the order to be rejected.

Branches and code coverage

Intended branches

- Change the status of the order.
  - ☐ Test coverage
- Transfer back the asset token.
  - ☐ Test coverage

Negative behavior

- The caller is the owner.

- ☐ Negative test
- Given order is the minting order.
  - ☐ Negative test
- Given order is pending.
  - ☐ Negative test

Function: `rejectRedeem(byte[32] orderHash)`

This function rejects the redemption order.

Inputs

- `orderHash`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be the order hash of the pending order.
  - **Impact**: Specifies the order to be rejected.

Branches and code coverage

Intended branches

- Change the status of the order.
  - ☐ Test coverage
- Transfer back the USSI token.
  - ☐ Test coverage

Negative behavior

- The caller is the owner.
  - ☐ Negative test
- Given order is the redemption order.
  - ☐ Negative test
- Given order is pending.
  - ☐ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to Ethereum Mainnet.

During our assessment on the scoped SSI Protocol contracts, we discovered seven findings. No critical issues were found. Three findings were of medium impact and four were of low impact.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.