code4rena

SSI Protocol

Smart Contract Security Assessment

Version 2.0

Audit dates: Dec 04 — Dec 13, 2024

Audited by: peakbolt
bin2chen

Contents

1. Introduction
   - 1.1 About Zenith
   - 1.2 Disclaimer
   - 1.3 Risk Classification
2. Executive Summary
   - 2.1 About SSI Protocol
   - 2.2 Scope
   - 2.3 Audit Timeline
   - 2.4 Issues Found
3. Findings Summary
4. Findings
   - 4.1 Medium Risk
   - 4.2 Low Risk
   - 4.3 Informational

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About SSI Protocol

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 Scope

Repository	SoSoValueLabs/ssi-protocol
Commit Hash	584f29cda02369b1483f7c5eefb2eb5634abb565
Mitigation Hash	4ff5f0db5951905f277d5e5a71025f0968102c06

2.3 Audit Timeline

DATE	EVENT
Dec 04, 2024	Audit start
Dec 13, 2024	Audit end
Dec 20, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	10
Informational	2
Total Issues	15

3. Findings Summary

ID	DESCRIPTION	STATUS
M-1	Missing pause function in `USSI`	Resolved

M-2	cancelSwapRequest() old orderInfo using new swap causing failure to cancel	Resolved
M-3	setIssuer() Missing change issuers[assetID] to newest	Resolved
L-1	Fee will be temporarily uncollectable when all assetTokens are burnt	Resolved
L-2	Lack of zero value check for `issueFee` could DoS minting/redeeming	Resolved
L-3	Lack of duplicate check in rebalance will cause baskets to be filled with duplicate tokens	Resolved
L-4	Lack of requester check for `orderHash` allows DoS of swap request	Resolved
L-5	Missing check for `chainId` in signature for `Order` and `HedgeOrder`	Resolved
L-6	setFeeManager() does not restrict burningFee() == false, which may cause the fee manager to not work properly.	Resolved
L-7	addMintRequest() user may pay fees more than expected	Resolved
L-8	addMintRequest() gives swap maximum allowance, may be a security risk	Resolved
L-9	when status is CANCEL reject redeem request , May result in failure to close the request	Resolved
L-10	Restrictions on outByContract can't rollback, which may result in the request not being closed properly.	Resolved
I-1	Redundant storage gap in `HedgeOrder`	Resolved
I-2	lock() It is recommended to use safeTransferFrom	Resolved

4. Findings

4.1 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] Missing pause function in `USSI`

Severity: Medium	Status: Resolved

Context:

- USSI.sol#L19

Description:

`USSI` inherits `PausableUpgradable` but does not implement functions to call `_pause()` and `_unpause()`.

This prevents the owner from pausing/unpausing the contract during an emergency.

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

Recommendation: Add `pause()` and `unpause()` to call the internal call `_pause()` and `_unpause()`.

SSI: Fixed in @2ac8fff8b32e..

Zenith: Resolved by adding `pause()` and `unpause()` with `onlyOwner` modifier in `USSI`.

[M-2] cancelSwapRequest() old orderInfo using new swap causing failure to cancel

Severity: Medium	Status: Resolved

Context:

- [AssetController.sol#L31]https://github.com/SyntheticAssets/ssi-protocol/blob/584f29cda02369b1483f7c5eefb2eb5634abb565/src/AssetController.sol#L31

Description:

Since `factory.swap` may have been modified by `AssetFactory.setSwap()`, `AssetIssuer.sol` will record the current `swap` when a new `request` is added.

```
    function addMintRequest(uint256 assetID, OrderInfo memory orderInfo)
external whenNotPaused returns (uint) {
    ...
        mintRequests.push(Request({
            nonce: mintRequests.length,
            requester: msg.sender,
            assetTokenAddress: assetTokenAddress,
            amount: order.outAmount,
@>          swapAddress: swapAddress,
            orderHash: orderInfo.orderHash,
            status: RequestStatus.PENDING,
            requestTimestamp: block.timestamp,
            issueFee: issueFee
        }));
        assetToken.lockIssue();
        emit AddMintRequest(mintRequests.length-1);
        return mintRequests.length-1;
    }
```

Subsequent `requests` that need to use swap take this value `request.swapAddress`.

But `AssetController.cancelSwapRequest/rollbackSwapRequest` doesn't do that and keeps using the latest `factoryAddress).swap()`.

This causes the old `orderInfo` to use the new `swap`, which prevents both methods from executing successfully.

Recommendation: If the owner can be trusted, it is recommended to add the parameter swap directly. If the owner is not trustworthy, it is recommended to pass nonce and abstract

method getSwap(nonce) to get the old swap.

SSI: Fixed with @2ac8fff8b32e... - Add the parameter `swap`

Zenith: Verified

[M-3] setIssuer() Missing change issuers[assetID] to newest

Severity: Medium	Status: Resolved

Context:

- AssetFactory.sol#L105

Description:

`AssetFactory.setIssuer()` is used to replace the `issuer` will perform the following two steps

1. revokeRole(old issuer)
2. grantRole(new issuer)

But currently the method does not update `issuers[assetID]`

```
    function setIssuer(uint256 assetID, address issuer) external
onlyOwner {
        require(issuer != address(0), "issuer is zero address");
        require(assetIDs.contains(assetID), "assetID not exists");
        IAssetToken assetToken = IAssetToken(assetTokens[assetID]);
        require(!assetToken.issuing(), "is issuing");
        address oldIssuer = issuers[assetID];
        assetToken.revokeRole(assetToken.ISSUER_ROLE(), oldIssuer);
        assetToken.grantRole(assetToken.ISSUER_ROLE(), issuer);
@>      // missing set issuers[assetID]
        emit SetIssuer(assetID, oldIssuer, issuer);
    }
```

This leads to two problems

1. the old `issuer` will not be de-authorized if it is modified again.
2. all uses of `AssetFactory.issuers()` will fail, e.g. `USSI.confirmMint()` to get the old `issuer`, resulting in the revert not executing properly.

note: `setRebalancer()/setFeeManager()` similar

Recommendation:

```
    function setIssuer(uint256 assetID, address issuer) external
onlyOwner {
...
       assetToken.revokeRole(assetToken.ISSUER_ROLE(), oldIssuer);
       assetToken.grantRole(assetToken.ISSUER_ROLE(), issuer);
```

```
+       issuers[assetID] = issuer;
        emit SetIssuer(assetID, oldIssuer, issuer);
```

SSI: @2ac8fff8b32e... - `setIssuer()/setRebalancer()/setFeeManager()` Are modified to update the latest.

Zenith: Verified

4.2 Low Risk

A total of 10 low risk findings were identified.

[L-1] Fee will be temporarily uncollectable when all assetTokens are burnt

Severity: Low	Status: Resolved

Context:

- AssetIssuer.sol#L336
- AssetIssuer.sol#L276

Description: Both `burnFor()` and `confirmRedeemRequest()` will call `assetToken.burn()`.

The issue is that when the last `burn()` reduces `totalSupply()` to `0`, it will prevent `collectFeeTokenset()` from collecting the fees as it only performs the fee collection when `totalSupply > 0`.

That means these fees will not be collectable till new AssetTokens are minted. If the AssetToken is to be deprecated, no new assetTokens will be minted and then it will require the fee collector to incur cost to mint AssetTokens to collect the fees.

```
    function burnFor(uint256 assetID, uint256 amount) external
whenNotPaused {
        IAssetFactory factory = IAssetFactory(factoryAddress);
        IAssetToken assetToken =
IAssetToken(factory.assetTokens(assetID));
        require(assetToken.allowance(msg.sender, address(this)) >=
amount, "not enough allowance");
        assetToken.lockIssue();
        assetToken.safeTransferFrom(msg.sender, address(this), amount);
        assetToken.burn(amount);
        assetToken.unlockIssue();
    }
```

Recommendation:

After `assetToken.burn()`, revert if `assetToken.feeCollected() == false` when `totalSupply() == 0`.

```
  function burnFor(uint256 assetID, uint256 amount) external
whenNotPaused {
        IAssetFactory factory = IAssetFactory(factoryAddress);
```

```
        IAssetToken assetToken =
IAssetToken(factory.assetTokens(assetID));
        require(assetToken.allowance(msg.sender, address(this)) >=
amount, "not enough allowance");
        assetToken.lockIssue();
        assetToken.safeTransferFrom(msg.sender, address(this), amount);
        assetToken.burn(amount);
+        // this reverts if feeCollected() == false  when totalSupply ==
0
+       require(assetToken.totalSupply() > 0 ||
assetToken.feeCollected(), "totalSupply() == 0but fee not collected");
        assetToken.unlockIssue();
    }
```

```
    function confirmRedeemRequest(uint nonce, OrderInfo memory orderInfo,
bytes[] memory inTxHashs) external onlyOwner {
        require(nonce < redeemRequests.length);
        Request memory redeemRequest = redeemRequests[nonce];
        checkRequestOrderInfo(redeemRequest, orderInfo);
        require(redeemRequest.status == RequestStatus.PENDING);
        ISwap swap = ISwap(redeemRequest.swapAddress);
        SwapRequest memory swapRequest =
swap.getSwapRequest(redeemRequest.orderHash);
        require(swapRequest.status == SwapRequestStatus.MAKER_CONFIRMED);
        swap.confirmSwapRequest(orderInfo, inTxHashs);
        IAssetToken assetToken =
IAssetToken(redeemRequest.assetTokenAddress);
        require(assetToken.balanceOf(address(this)) >=
redeemRequest.amount, "not enough asset token to burn");
        Order memory order = orderInfo.order;
        Token[] memory outTokenset = order.outTokenset;
        address vault = IAssetFactory(factoryAddress).vault();
        for (uint i = 0; i < outTokenset.length; i++) {
            address tokenAddress =
Utils.stringToAddress(outTokenset[i].addr);
            IERC20 outToken = IERC20(tokenAddress);
            uint outTokenAmount = outTokenset[i].amount * order.outAmount
/ 10**8;
            uint feeTokenAmount = outTokenAmount * redeemRequest.issueFee
/ 10**feeDecimals;
            uint transferAmount = outTokenAmount - feeTokenAmount;
            require(outToken.balanceOf(address(this)) >= outTokenAmount,
"not enough balance");
            outToken.safeTransfer(redeemRequest.requester,
transferAmount);
```

```
            outToken.safeTransfer(vault, feeTokenAmount);
        }
        assetToken.burn(redeemRequest.amount);
+        // this reverts if feeCollected() == false  when totalSupply ==
0
+       require(assetToken.totalSupply() > 0 ||
assetToken.feeCollected(), "totalSupply() == 0 but fee not collected");
        redeemRequests[nonce].status = RequestStatus.CONFIRMED;
        assetToken.unlockIssue();
        emit ConfirmRedeemRequest(nonce);
    }
```

SSI: Fixed in [@2ac8fff8b32e..](). It's acceptable that we couldn't collect fee if the totalSupply is zero, and we also block collect fee while issuing. But we will check feeCollected in burnFor.

Zenith: Verified! Resolved by checking in `burnFor()` and revert when `feeCollect() == false`.

[L-2] Lack of zero value check for `issueFee` could DoS minting/redeeming

Severity: Low	Status: Resolved

Context:

- AssetIssuer.sol#L175-L177

Description:

Since issueFee can be zero, it is recommended to check that `feeTokenAmount > 0` before calling `inToken.safeTransfer(vault, feeTokenAmount);`. That is because some ERC token like LEND will revert on zero transfer.

```
            uint feeTokenAmount = inTokenAmount * mintRequest.issueFee /
10**feeDecimals;
            require(inToken.balanceOf(address(this)) >= feeTokenAmount,
"not enough balance");
            inToken.safeTransfer(vault, feeTokenAmount);
```

Recommendation:

```
+ if(feeTokenAmount > 0)
      inToken.safeTransfer(vault, feeTokenAmount);
```

SSI: Fixed in @2ac8fff8b32e...

Zenith: Verified

[L-3] Lack of duplicate check in rebalance will cause baskets to be filled with duplicate tokens

Severity: Low	Status: Resolved

Context:

- AssetRebalancer.sol#L44-L48

Description: There are no duplicate checks for `Utils.addTokenset()` so its possible to have duplicate token in the `inBasket` via `orderInfo.order.outTokenset`, which will cause it to call `assetToken.rebalance()` during `confirmBalanceRequest()` with a new basket/tokenset with duplicate tokens.

Duplicate tokens in the basket is undesirable as the assetToken mint and burn functions assumes that the basket's tokens are unique. This issue could cause the tokenset operations (add/sub) to incorrectly update the tokens amount.

This also applies to `addBurnFeeRequest()`.

```
    function addRebalanceRequest(uint256 assetID, Token[] memory basket,
OrderInfo memory orderInfo) external onlyOwner returns (uint256) {
        IAssetFactory factory = IAssetFactory(factoryAddress);
        address assetTokenAddress = factory.assetTokens(assetID);
        IAssetToken assetToken = IAssetToken(assetTokenAddress);
        address swapAddress = factory.swap();
        ISwap swap = ISwap(swapAddress);
        require(assetToken.totalSupply() > 0, "zero supply");
        require(assetToken.feeCollected(), "has fee not collect");
        require(assetToken.hasRole(assetToken.REBALANCER_ROLE(),
address(this)), "not a rebalancer");
        require(assetToken.rebalancing() == false, "is rebalancing");
        require(assetToken.issuing() == false, "is issuing");
        require(swap.checkOrderInfo(orderInfo) == 0, "order not valid");
        require(keccak256(abi.encode(assetToken.getBasket())) ==
keccak256(abi.encode(basket)), "underlying basket not match");
        Token[] memory inBasket =
Utils.muldivTokenset(orderInfo.order.outTokenset,
orderInfo.order.outAmount, 10**8);
        Token[] memory outBasket =
Utils.muldivTokenset(orderInfo.order.inTokenset,
orderInfo.order.inAmount, 10**8);
        require(Utils.containTokenset(basket, outBasket), "not enough
balance to sell");
```

```
        Token[] memory newBasket =
Utils.addTokenset(Utils.subTokenset(basket, outBasket), inBasket);
        Token[] memory newTokenset = Utils.muldivTokenset(newBasket,
10**assetToken.decimals(), assetToken.totalSupply());
        for (uint i = 0; i < newTokenset.length; i++) {
            require(newTokenset[i].amount > 0, "too little left in new
basket");
        }
```

SSI: Will check duplicate tokens in Swap.checkOrderInfo. Considering the gas cost, only check duplicate tokens in AssetRebalancer. In addBurnFeeRequest, no need to check duplicate tokens because it doesn't update the tokenset of asset token.

Fixed in @2ac8fff8b32e...

Zenith: Verified. Resolved by checking in `addRebalanceRequest()` to ensure the `newTokenset()` does not have duplicate tokens.

[L-4] Lack of requester check for `orderHash` allows DoS of swap request

Severity: Low	Status: Resolved

Description:

As `orderHash` is not tied to a specific user, someone else can use that user's `orderHash` for another request, which then adds it to `orderHashs`, causing the user's request to fail the check as it has been added the `orderHashs`.

An attacker can exploit this issue by frontrunning the victim's request using the victim's `orderHash`, causing the victim's request to fail as it would have been added to `orderHashs` by the attacker.

```
struct Order {
    address maker;
    uint256 nonce;
    Token[] inTokenset;
    Token[] outTokenset;
    string[] inAddressList;
    string[] outAddressList;
    uint256 inAmount;
    uint256 outAmount;
    uint256 deadline;
}

struct OrderInfo {
    Order order;
    bytes32 orderHash;
    bytes orderSign;
}
```

Recommendation: Suggest to only allow use of `orderHash` for a specific requester by adding it to the `Order` struct and validate it in `checkOrderInfo()`.

SSI: Fixed in @2ac8fff8b32e.. We will check requester == msg.sender in AssetIssuer.

Zenith: Resolved by checking `orderInfo.order.requester == msg.sender` in `addMintRequest()` and `addRedeemRequest()`.

[L-5] Missing check for `chainId` in signature for `Order` and `HedgeOrder`

Severity: Low	Status: Resolved

Context:

- Swap.sol#L57-L58
- USSI.sol#L137

Description: In both `USSI` and `Swap`, there is no chainId in the signed hash.

This will allow users to re-use the same signature that was meant for one chain and replay it on another chain. When that happens, the maker and issuer will need to verify the re-played transaction and reject them. In the worst case it could be used to prevent rebalancing by doing a swap to `lockIssue()`.

```
    struct HedgeOrder {
        HedgeOrderType orderType;
        uint256 assetID;
        address redeemToken;
        uint256 nonce;
        uint256 inAmount;
        uint256 outAmount;
        uint256 deadline;
        address requester;
        uint256[5] __gap;
    }

    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");
    }
```

```
    struct Order {
        address maker;
        uint256 nonce;
        Token[] inTokenset;
        Token[] outTokenset;
        string[] inAddressList;
        string[] outAddressList;
        uint256 inAmount;
        uint256 outAmount;
        uint256 deadline;
    }

    struct OrderInfo {
        Order order;
        bytes32 orderHash;
        bytes orderSign;
    }

    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;
    }
```

Recommendation: If there is an intention to deploy on multiple chains, it is recommended to add `chainId` to the `Order` struct for signing and verify the signature in `checkOrderInfo()` that it is intended for the specific `block.chainid` of swap contract.

SSI: Fixed in @2ac8fff8b32e..., @7e3adbfc8f4e.. and @4ff5f0db59519..

Zenith: Verified & resolved by adding checks in both `Swap.checkOrderInfo()` and `USSI.checkHedgeOrder()` to ensure that the chain in the order matches the chain specified in the contract to prevent re-use of order signature across chains.

The fix introduced a new issue - chain is added to the beginning of the struct for HedgeOrder and Order. This will load the wrong values and override the existing values of orders if you are upgrading, which was then mitigated with @4ff5f0db59519..

[L-6] setFeeManager() does not restrict burningFee() == false, which may cause the fee manager to not work properly.

Severity: Low	Status: Resolved

Context:

- AssetFactory.sol#L127

Description: `setFeeManager()` is used to set the new `FeeManager`, but currently there is no restriction `burningFee() == false`.

If the old one is doing a burn fee request, the switch will cause old FeeManager and new FeeManager both not work properly

1. old FeeManager `addBurnFeeRequest()` => change: `assetToken.lockBurnFee = true`
2. owner change fee manager
3. old FeeManager `confirmBurnFeeRequest()` => revert , not permission
4. new FeeManager `addBurnFeeRequest()` => revert , `assetToken.lockBurnFee = true` can't add request

Recommendation:

```
    function setFeeManager(uint256 assetID, address feeManager) external
onlyOwner {
        require(feeManager != address(0), "feeManager is zero address");
        require(assetIDs.contains(assetID), "assetID not exists");
        IAssetToken assetToken = IAssetToken(assetTokens[assetID]);
+       require(assetToken.burningFee() == false, "is burning fee");
```

SSI: Fixed with @2ac8fff8b32e... - limit `assetToken.burningFee()==false`

Zenith: Verified

[L-7] addMintRequest() user may pay fees more than expected

Severity: Low	Status: Resolved

Context

- AssetIssuer.sol#L103

Description: `addMintRequest()` doesn't have a maximum fee limit, if the fee is adjusted by the `owner` to become larger, the user may pay a larger fee than expected, resulting in losses

The participant cost includes the fee: `feeTokenAmount = inTokenset[i].amount * order.inAmount * _issueFees[assetID]` Where `_issueFees[assetID]`, the owner can make adjustments, and it is effective immediately

This fee may exceed the user's expectations

Example:

1. participant see fees= 0.01%,and submits request.
2. owner modifies it to 0.02% , before the request transaction is executed
3. participant transaction executes with 1x more fee than expected, which is not too low if order.inAmount is large enough.

Note: `addRedeemRequest()` similar

Recommendation: Add parameter max_fees to limit the maximum acceptable fee.

SSI: @2ac8fff8b32e... - `addMintRequest()/addRedeemRequest()` Add parameter `maxIssueFee` to limit the maximum fee.

Zenith: Verified

[L-8] addMintRequest() gives swap maximum allowance, may be a security risk

Severity: Low	Status: Resolved

Context:

- AssetIssuer.sol#L108

Description: `addMintRequest()` will give `swap` maximum allowance, `type(uint256).max` but this swap is variable and the old one will still have unused allowance. There is a security risk

```
    function setSwap(address swap_) external onlyOwner {
        require(swap_ != address(0), "swap address is zero");
        swap = swap_;
        emit SetSwap(swap);
    }
```

```
    function addMintRequest(uint256 assetID, OrderInfo memory orderInfo)
external whenNotPaused returns (uint) {
    ...
        for (uint i = 0; i < inTokenset.length; i++) {
            require(bytes32(bytes(inTokenset[i].chain)) ==
bytes32(bytes(factory.chain())), "chain not match");
            address tokenAddress =
Utils.stringToAddress(inTokenset[i].addr);
            IERC20 inToken = IERC20(tokenAddress);
            uint inTokenAmount = inTokenset[i].amount * order.inAmount /
10**8;
            uint feeTokenAmount = inTokenAmount * issueFee /
10**feeDecimals;
            uint transferAmount = inTokenAmount + feeTokenAmount;
            require(inToken.balanceOf(msg.sender) >= transferAmount, "not
enough balance");
            require(inToken.allowance(msg.sender, address(this)) >=
transferAmount, "not enough allowance");
            if (inToken.allowance(address(this), swapAddress) <
inTokenAmount) {
@>              inToken.forceApprove(swapAddress, type(uint256).max);
            }
            inToken.safeTransferFrom(msg.sender, address(this),
transferAmount);
        }
```

Recommendation: It is recommended to remove the allowance setting in `addMintRequest()`. In `AssetIssuer.confirmMintRequest()` give the allowance when it is really needed, and only give the needed amount of `orderInfo.inTokenset[i].amount * order.inAmount / 10**8`, and keep the allowance to 0 after use.

SSI: Fixed with [@7e3adbfc8f4e...]

1. in `addMintRequest()` remove approve allowance
2. in `AssetIssuer.confirmMintRequest()` give the allowance when it is really needed, and only give the needed amount of `orderInfo.inTokenset[i].amount * order.inAmount / 10**8`, and keep the allowance to 0 after use.

Zenith: Verified

[L-9] when status is CANCEL reject redeem request , May result in failure to close the request

Severity: Low	Status: Resolved

Context:

- AssetIssuer.sol#L243

Description: Currently `rejectRedeemRequest()`, which only allows execution when `swapRequest.status == SwapRequestStatus.REJECTED`

```
    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");
```

`swapRequest.status == SwapRequestStatus.CANCEL` does not allow `reject redeem request`

May result in failure to close request

The following scenario may be a problem

1. bob call addRedeemRequest(maker = alice) -> assetToken.lockIssue()
2. alice can't confirm (lost key or Malicious)
3. owner call cancelSwapRequest()
4. owner call rejectRedeemRequest() => revert
5. assetToken still lockIssue

Recommendation:

```
    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");
+       require(swapRequest.status == SwapRequestStatus.REJECTED ||
swapRequest.status == SwapRequestStatus.CANCEL);
```

SSI: @2ac8fff8b32e... - allow `swapRequest.status == SwapRequestStatus.CANCEL`

Zenith: Verified

[L-10] Restrictions on outByContract can't rollback, which may result in the request not being closed properly.

Severity: Low	Status: Resolved

Context.

- Swap.sol#L187

Description: When executing `rollbackSwapRequest()`, the current protocol restricts `require(!!swapRequests[orderHash].outByContract)`, it could lead to a malicious participant + maker locking assetToken.lockIssue.

Example:

1. bob call addRedeemRequest() -> assetToken.lockIssue() with fake outToken
2. alice call `swap.makerConfirmSwapRequest()` with fake outToken
3. alice bob disappear
4. owner call rollbackSwapRequest() => revert , can't be `outByContract`
5. owner call `AssetIssuer.confirmRedeemRequest()` => revert when fake `outToken.safeTransfer()`
6. assetToken will lockIssue

We can work around this by `owner call AssetFactory.setIssuer(jack)=> jack call AssetToken.unlockIssue()`. But it's a bit weird.

Recommendation: Adding method like: `forceRollbackSwapRequest() external onlyRole(DEFAULT_ADMIN_ROLE)`

Note: A small modification suggestion

`AssetIssuer.sol`

```
    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)));
            address tokenAddress = tokenAddresses[i];
            if (tokenAddress != address(0)) {
                IERC20 token = IERC20(tokenAddress);
-               if (token.balanceOf(address(this)) >
tokenClaimables[tokenAddress]) {
+               if (token.balanceOf(address(this)) >=
tokenClaimables[tokenAddress]) {
                    token.safeTransfer(owner(),
token.balanceOf(address(this)) - tokenClaimables[tokenAddress]);
                }
            }
        }
    }
```

SSI: Fixed with [@2ac8fff8b32e4...] - Add `SwapRequestStatus.FORCE_CANCEL`and add the `AssetIssuer.claim()` mechanism.

Zenith: Verified

4.3 Informational

A total of 2 informational findings were identified.

[I-1] Redundant storage gap in `HedgeOrder`

Severity: Informational	Status: Resolved

Context:

- USSI.sol#L36

Description: As the struct `HedgeOrder` is only used in the mapping `hedgeOrders`., it does not need a storage `__gap` as mapping do not store values contiguously like normal storage variables. That means the the values in the struct will not be overridden by the next element in the mapping.

What should be ensured is that the new data types in the struct should only be added at the end and existing data types should not be removed.

```
struct HedgeOrder {
    HedgeOrderType orderType;
    uint256 assetID;
    address redeemToken;
    uint256 nonce;
    uint256 inAmount;
    uint256 outAmount;
    uint256 deadline;
    address requester;
    uint256[5] __gap;
}
```

Recommendation: Remove `__gap` from `HedgeOrder`.

SSI: Fixed in @2ac8fff8b32e...

Zenith: Verified. Noted that `receiver` is added to `HedgeOrder` to faciliate off-chain transfer.

[I-2] lock() It is recommended to use safeTransferFrom

Severity: Informational	Status: Resolved

Context:

- AssetLocking.sol#L109

Description: `lock()` Using `IERC20(token).transferFrom()` may fail for some irregular `erc20`. Some `erc20`, `transferFrom` method signatures do not have a `return bool`. These `tokens` may cause `lock()` to fail.

Recommendation. Use `SafeERC20.safeTransferFrom`.

SSI: @2ac8fff8b32e...

Zenith: Verified