Set Protocol Exchange Issuance Smart Contract Audit

1. Introduction

iosiro was commissioned by Set Protocol to conduct a smart contract audit of the community developed Exchange Issuance functionality for Set Protocol V2. The audit was conducted between 15 and 18 February with two auditors, consuming a total of 8 person-days. A 2 person-day review of remediations was conducted by one auditor on 8 and 9 March. Finally, changes made to remediate new issues raised in the review were reviewed on 17 March.

This report is organized into the following sections.

- **Section 2 - Executive summary:** A high-level description of the findings of the audit.
- **Section 3 - Audit details:** A description of the scope and methodology of the audit.

- **Section 4 - Design specification:** An outline of the intended functionality of the smart contracts.
- **Section 5 - Detailed findings:** Detailed descriptions of the findings of the audit.

The information in this report should be used to better understand the risk exposure of the smart contracts, and as a guide to improving the security posture of the smart contracts by remediating issues identified. The results of this audit are only a reflection of the source code reviewed at the time of the audit and of the source code that was determined to be in-scope.

The purpose of this audit was to achieve the following:

- Identify potential security flaws.
- Ensure that the smart contracts functioned according to the documentation provided.

Assessing the economics, game theory, or underlying business model of the platform were strictly beyond the scope of this audit.

Due to the unregulated nature and ease of transfer of cryptocurrencies, operations that store or interact with these assets are considered very high risk with regards to cyber attacks. As such, the highest level of security should be observed when interacting with these assets. This requires a forward-thinking approach, which takes into account the new and experimental nature of blockchain technologies. Strategies that should be used to encourage secure code development include:

- Security should be integrated into the development lifecycle and the level of perceived security should not be limited to a single code audit.
- Defensive programming should be employed to account for unforeseen circumstances.
- Current best practices should be followed where possible.

2. Executive summary

This report presents the findings of an audit performed by iosiro on Set Protocol's Exchange Issuance smart contracts. The Exchange Issuance product contract extends

the existing Basic Issuance functionality and allows users to issue and redeem Sets using Ether or a single ERC-20 token, by abstracting the exchange of tokens on decentralized exchanges (DEX).

Audit findings

During the audit, iosiro identified three medium risk issues and one low risk issue, as well as several informational issues. All medium and low risk, as well as the majority of informational issues, were remediated following the initial audit, and issues introduced during this process were remediated following the initial review. Three design comments relating to refactoring, spelling mistakes and storage considerations were partially addressed but remain open.

Overall, the code was found to be of a high standard and primarily followed best practices.

Further recommendations

At a high level, the security posture of the Exchange Issuance smart contracts could be further strengthened by conducting regular audits as functionality is added or changed.

3. Audit details

3.1 Scope

The source code considered in-scope for the assessment is described below. Code from all other files is considered to be out-of-scope. Out-of-scope code that interacts with in-scope code is assumed to function as intended and introduce no functional or security vulnerabilities for the purposes of this audit.

3.1.1 Set smart contracts

Project name: index-coop-contracts
Initial audit commit: a2904cf

Final review audit commit: 16516ad

Files: ExchangeIssuance.sol

Note: commits following the initial audit commit have only been reviewed insofar as they relate to audit remediations.

3.2 Methodology

A variety of techniques were used while conducting the audit. These techniques are briefly described below.

3.2.1 Code review

The source code was manually inspected to identify potential security flaws. Code review is a useful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high risk areas of the system.

3.2.2 Dynamic analysis

The contracts were compiled, deployed, and tested in a Ganache test environment. Manual analysis was used to confirm that the code operated at a functional level, and to verify the exploitability of any potential security issues identified. The coverage report of the provided tests is given below.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
adapters/	100	96.05	100	99.43	
FeeSplitAdapter.sol	100	83.33	100	100	
FlexibleLeverageStrategyAdapter.sol	100	97.14	100	99.32	906
exchangeIssuance/	100	100	100	100	
ExchangeIssuance.sol	100	100	100	100	
hooks/	100	100	75	100	
SupplyCapIssuanceHook.sol	100	100	75	100	
lib/	50.91	48.08	58.33	54.31	
AddressArrayUtils.sol	28.57	28.57	42.86	29.27	... 147,148,150
BaseAdapter.sol	100	78.57	100	100	

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
MutualUpgrade.sol	100	100	100	100	
PreciseUnitMath.sol	14.29	0	26.67	14.29	... 182,185,188
TimeLockUpgrade.sol	100	75	100	100	
manager/	100	100	100	100	
BaseManager.sol	100	100	100	100	
ICManager.sol	100	100	100	100	
staking/	0	0	0	0	
RewardsDistributionRecipient.sol	0	0	0	0	9,10
StakingRewards.sol	0	0	0	0	... 137,138,140
StakingRewardsV2.sol	0	0	0	0	... 140,141,143
token/	28.35	18.33	46.15	28.35	
IndexToken.sol	21.74	14.29	42.11	21.74	... 297,298,299
MerkleDistributor.sol	100	83.33	100	100	
Vesting.sol	0	0	0	0	... 52,54,55,57
All files	69.97	66.15	71.36	70.31	

3.2.3 Automated analysis

Tools were used to automatically detect the presence of several types of security vulnerabilities, including reentrancy, timestamp dependency bugs, and transaction-ordering dependency bugs. The static analysis results were manually analysed to remove false positive results. True positive results would be indicated in this report. Static analysis tools commonly used include Slither, MythX, as well as Securify. Furthermore, the Remix IDE, compilation output, and linters are also used to identify potential areas of concern.

3.3 Risk ratings

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

- **High risk** - The issue could result in a loss of funds for the contract owner or system users.
- **Medium risk** - The issue resulted in the code specification being implemented incorrectly.
- **Low risk** - A best practice or design issue that could affect the security of the contract.
- **Informational** - A lapse in best practice or a suboptimal design pattern that has a minimal risk of affecting the security of the contract.
- **Closed** - The issue was identified during the audit and has since been addressed to a satisfactory level to remove the risk that it posed.

4. Design specification

The following section outlines the intended functionality of the system at a high level. The information is based on internal documentation provided by the Set Protocol team.

System description

`ExchangeIssuance` is a product contract, not a module extension, that allows a user the ability to issue and redeem a Set using Ether or a single ERC-20 token and DEX liquidity. The user provides said token (e.g. ETH, USDC, USDT), purchases each of a Set's constituent tokens via Uniswap or SushiSwap, and calls Set's `BasicIssuanceModule` to mint the Set.

This contract also houses the logic for redemption which is the reverse process, taking in a SetToken quantity, redeeming it via the `BasicIssuanceModule`, and selling each constituent for some token on Uniswap or SushiSwap.

Function descriptions

Name	Description
`constructor`	Sets state and approves WETH transfers to Uniswap and SushiSwap.

Name	Description
approveToken	Runs all the necessary approval functions required for a given ERC-20 token. This function can be called when a new token is added to a SetToken during a rebalance.
approveTokens	Runs all the necessary approval functions required for a list of ERC-20 tokens.
approveSetToken	Runs all the necessary approval functions required before issuing or redeeming a SetToken. This function needs to be called only once before the first time this smart contract is used on any particular SetToken.
issueSetForExactToken	Issues SetTokens for an exact amount of input ERC-20 tokens. The ERC-20 token must be approved by the sender to this contract.
issueSetForExactETH	Issues SetTokens for an exact amount of input ether.
issueExactSetFromToken	Issues an exact amount of SetTokens for a given amount of input ERC-20 tokens.
issueExactSetFromETH	Issues an exact amount of SetTokens using a given amount of ether.
redeemExactSetForToken	Redeems an exact amount of SetTokens for an ERC-20 token. The SetToken must be approved by the sender to this contract.
redeemExactSetForETH	Redeems an exact amount of SetTokens for ETH. The SetToken must be approved by the sender to this contract.
getEstimatedIssueSetAmount	Returns an estimated quantity of the specified SetToken given a specified amount of input token. Estimating pulls the best price of each component using UniSwap or Sushiswap.
getAmountInToIssueExactSet	Returns the amount of input tokens required to issue an exact amount of SetTokens.
getEstimatedRedeemSetAmount	Returns an estimated amount of ETH or specified ERC-20 received for a given SetToken and SetToken amount.

5. Detailed findings

The following section includes in depth descriptions of the findings of the audit.

5.1 High risk

No high risk issues were present at the conclusion of the audit.

5.2 Medium risk

No medium risk issues were present at the conclusion of the audit.

5.3 Low risk

No low risk issues were present at the conclusion of the audit.

5.4 Informational

5.4.1 Design comments

Actions to improve the functionality and readability of the codebase are outlined below.

Refactoring suggestions

Portions of the code can be refactored to improve readability and consistency, as indicated below.

1. ExchangeIssuance.sol#L363, ExchangeIssuance.sol#L549: The revert message should be updated to indicate that the output amount may have been insufficient, or that the pair does not exist on the corresponding exchange.

Fix typos, spelling and grammatical errors

Spelling mistakes were identified in the codebase. Fixing these mistakes can help improve the end-user experience by providing clear information on errors encountered, and improve the maintainability and auditability of the codebase.

1. ExchangeIssuance.sol#L519: `allownce -> allowance`
2. ExchangeIssuance.sol#L91: `redunded -> refunded`

Storage considerations

The state variables of the `exchangeIssuance` contract are addresses that are meant to be constant throughout the lifetime of the contract. As such, these values should be marked as `immutable`. Immutable state variables save storage slots, which in turn reduce gas fees for reading.

Further reading on immutables can be found here.

Update

While the majority of value-type state variables were made immutable in 8417960, `address public WETH` remained mutable.

5.5 Closed

5.5.1 Sets with illiquid components revert when issuing (medium risk)

ExchangeIssuance.sol#L731

Description

When issuing Sets, the required components are acquired through either Uniswap or SushiSwap. If a component exists on both exchanges, but does not have sufficient liquidity on either exchange, the transaction will revert due to a SafeMath subtraction overflow.

This issue was mitigated as the Set can still be issued through the Basic Issuance Module, provided the ERC-20 tokens are acquired elsewhere.

Recommendation

Try-catch statements should be used to react if the call to the `getAmountIn()` function reverts for the Uniswap or SushiSwap libraries. In this way, if there is at least sufficient liquidity on either exchange, the transaction can still proceed.

Update

This was remediated in 8417960 by the addition of `if` and `require` statements in `_getMaxTokenForExactToken(...)` and `_getMinTokenForExactToken(...)`. This avoided the subtraction overflow and allowed the transaction to revert with an appropriate error message in the event of illiquidity.

5.5.2 Sets with an ETH or WETH component cannot be issued or redeemed (medium risk)

ExchangeIssuance.sol#L731

Description

If a user purchases a Set with ETH, the ETH is wrapped to become WETH before acquiring the Set's components through Uniswap or Sushiswap. However, as there is no `WETH/WETH` liquidity pair in either Uniswap or SushiSwap, the `_getMinTokenForExactToken()` function returns `PreciseUnitMath.maxUint256()` as the minimum amount of WETH required. This causes addition overflows in subsequent functions, causing the transaction to revert. This result also occurs if a user purchases Sets using WETH as an ERC-20 token directly.

Similarly, Sets with WETH components cannot be redeemed as there is a requirement in the `sortTokens()` function from the UniSwap library, or SushiSwap library, that requires trades to be between two different tokens.

Recommendation

Any time a Set with a WETH component is used, additional validation should take place before the corresponding Uniswap or SushiSwap trade. As an example for issuance, `_getMinTokenForExactToken()` should check if `_tokenB` is WETH, and if it is, return the `_amountOut`. As an example for redemption, the `_liquidateComponentsForWETH()` function should validate that the component is not WETH before performing any additional actions.

Update

This was remediated in 8417960 by adding an `if` statement that returned the original amount in the event that both input and output tokens were the same. This `if` statement was added to the following functions:

- `_getMinTokenForExactToken(...)`
- `_getMaxTokenForExactToken(...)`
- `_swapExactTokensForTokens(...)`
- `_swapTokensForExactTokens(...)`

5.5.3 Incorrect parameters passed to `_getMaxTokenForExactToken` (medium risk)

ExchangeIssuance.sol#L380

Description

When calling the `getEstimatedIssueSetAmount()` function, the max amount of WETH from the given `_inputToken` should be calculated with the `_getMaxTokenForExactToken()` function. However, the function parameters were incorrectly switched around, which instead calculated the max amount of `_inputToken` for the same amount of WETH.

While this estimate would likely affect the dApp interfacing with the `exchangeIssuance` contract, the risk of this issue was limited due to it being a view function.

Recommendation

The correct calculation can be performed with, `_getMaxTokenForExactToken(_amountInput, address(_inputToken), address(WETH))`, which fetches the max amount of input token for the `_amountInput` of WETH.

Update

This was remediated in 8417960 according to the recommendation above.

5.5.4 Incorrect `else if` statement (medium risk)

ExchangeIssuance.sol#L465

Description

While remediating findings from the initial audit, a new bug was introduced through an incorrect `if else` statement in the `getEstimatedIssueSetAmount(...)` function, which would prevent the function from working correctly for components using the SushiSwap exchange. This is shown in the fourth line of the code extract below:

```
if (exchanges[i] == Exchange.Uniswap) {
    (uint256 reserveIn, uint256 reserveOut) = UniswapV2Library.getReserves(uniFactory,
    amountTokenOut = UniswapV2Library.getAmountOut(scaledAmountEth, reserveIn, reserve
} else if (exchanges[i] == Exchange.Uniswap) {
    (uint256 reserveIn, uint256 reserveOut) = SushiswapV2Library.getReserves(sushiFact
    amountTokenOut = SushiswapV2Library.getAmountOut(scaledAmountEth, reserveIn, reser
}
```

Recommendation

The value `Exchange.Uniswap` in the `if else` statement should be changed to `Exchange.Sushiswap`.

Update

The recommendation above was implemented in 2590cff.

5.5.5 Insufficient test quality and coverage (low risk)

ExchangeIssuance.sol

Description

Certain functionality was found to have insufficient unit test quality and coverage. This could impact the maintainability of the code, and increase the likelihood of introducing functional or security issues into the codebase. The following functionality was not covered by tests:

- Logic for dealing with a WETH output token in the external function `redeemExactSetToken(...)`.
- Logic for using SushiSwap in the external function `getEstimatedIssueSetAmount(...)` and internal functions `_getMinTokenForExactToken(...)` and `_getMaxTokenForExactToken(...)`.

A full coverage report can be found in 3.2.2 Dynamic analysis above.

Recommendation

The test suite should be extended to cover the uncovered functionality.

Update

The test suite was extended to cover this functionality in PR-13.

5.5.6 Use of `transfer` function (low risk)

ExchangeIssuance.sol#L247, *ExchangeIssuance.sol#L279*, *ExchangeIssuance.sol#L349*

Description

The `issueExactSetFromToken`, `issueExactSetFromETH` and `redeemExactSetForETH` functions made use of the `transfer()` function to send ether. While `transfer()` is commonly used to prevent reentrancy attacks due to its 2300 gas limit, it relies on the receiving contract having a fallback function below this limit. As demonstrated in EIP-1884, which changed the gas cost of the `SLOAD` operation, gas costs can change. This could lead to a case where a contract has its fallback function increased above the 2300 limit, resulting in it becoming incompatible with the system. More information can be found in Consensys On Avoiding `transfer()`.

Recommendation

It is recommended that the `call()` function be used to send ether instead of `transfer(...)`. Alternatively, the `sendValue(...)` function from the OpenZeppelin Address library could be used.

Update

This was remediated in 8417960 by replacing all `transfer(...)` calls with `sendValue(...)` calls.

5.5.7 Unrestricted fallback function (informational)

ExchangeIssuance.sol#L355

Description

The fallback function was unrestricted, which allowed anyone to send ETH directly to the ExchangeIssuance contract. As the contract makes use of its entire balance between issuing and redeeming its Sets, users who send ETH to the contract through the fallback function are at risk of having their funds used by other users. Restricting the fallback function will add a safeguard in the case of users accidentally sending their ETH to the contract directly.

Recommendation

The `receive` function should validate that `msg.sender` is the WETH contract to ensure that users do not accidentally deposit ETH funds in the contract.

Update

This was remediated in 8417960 according to the recommendation above.

5.5.8 `approveSetToken()` missing the `isSetToken()` modifier (informational)

ExchangeIssuance.sol#L152

Description

The `approveSetToken()` function takes in a Set and approves the exchange routers and the Basic Issuance Module with an allowance. However, the address passed was not validated to be a Set, and as such, any address could be used to approve an allowance for the contract.

As any ERC-20 token could be approved through the `approveToken()` function, this issue is raised as informational to be consistent with the other functions that make use of Sets.

Recommendation

The `isSetToken()` modifier should be added to the `approveSetToken()` function to ensure that the address passed in is a valid Set.

Update

This was remediated in 8417960 according to the recommendation above.

5.5.9 Design comments (informational)

Actions to improve the functionality and readability of the codebase are outlined below.

Refactoring suggestions

Portions of the code can be refactored to improve readability and consistency, as indicated below.

1. ExchangeIssuance.sol#L222: The description of `_maxAmountInputToken` in the `issueExactSetFromToken()` function should be updated to make it clear that the full amount of input tokens will be converted to WETH and refunded as ETH, not as the `_inputToken`.
2. ExchangeIssuance.sol#L399: A revert message should be included to indicate the reason for the transaction reverting. An example of an appropriate message would be "Exchange not supported".
3. ExchangeIssuance.sol#L540: The `_issueSetForExactWETH` function can be renamed to `_issueSetForWETH` since the entire contract balance of WETH is used. While this should always be the WETH as sent by the user, it may be incorrect in situations where there is additional WETH as sent by another user in the case of error.

Update

All refactoring suggestions above were addressed in 8417960.

1. The parameter description was clarified.
2. After this was addressed in 8417960, the affected `require` statement was refactored into an else-if condition in 17e90f0.
3. This function was changed to better match the behavior implied by its name (`_issueSetForExactWETH`).

Gas optimization

1. ExchangeIssuance.sol#L275: The `issueExactSetFromETH()` function can revert if the `amountEth` used from issuing some Set is more than the `msg.value`. This may happen if the price of the components is higher than expected by the time the

transaction is processed. It may be feasible to get the expected rates from Uniswap or SushiSwap of the expected components in order to prevent a revert further in the transaction.

Update

A check to prevent the revert was added in ExchangeIssuance.sol#603 in 8417960.

Test quality

While the test coverage of the `exchangeIssuance` contract was near complete, several test cases had loose requirements for passing. In particular, the output of certain values just needed to be more or less than an expected value. This practice is less robust than its counterpart of testing for exact values (or almost exact where appropriate), which as an example, lead to the Incorrect parameters passed to `_getMaxTokenForExactToken` issue. Test cases should be reworked to have stricter pass requirements, indicating that the functions are working as intended.

Update

The test cases were reworked in 17e90f0.

Use of contract balances

The contract relied on its own ETH and ERC-20 balances when issuing and redeeming Sets. While there is no inherent risk in doing this provided that transactions are accounted for correctly, it does mean that any ETH or ERC-20 tokens sent to the contract erroneously or otherwise is subject to theft by other users. If there is any remaining balance of ETH or ERC-20, a user may simply receive more of a given Set when transacting with the contract.

Update

Functions changed to use input amounts instead of contract balances in 8417960 and 23dc3e2.

Fix typos, spelling and grammatical errors

Spelling mistakes were identified in the codebase. Fixing these mistakes can help improve the end-user experience by providing clear information on errors encountered, and improve the maintainability and auditability of the codebase.

1. ExchangeIssuance.sol#L85: `addres` -> `address`

2. ExchangeIssuance.sol#L543: `aproval -> approval`

Update

These spelling mistakes were fixed in d207aaf.