Synthetix Phase 2 Smart Contract Audit

1. Introduction

iosiro was commissioned by Synthetix to conduct a smart contract audit on portions of the Synthetix stablecoin system. The audit was performed between 14 August 2019 and 25 September 2019.

This report is organised 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 understand the risk exposure of the smart contracts, and as a guide to improve the security posture of the smart contracts by remediating the issues that were 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.

2. Executive Summary

This report presents the findings of a second audit performed by iosiro on the Synthetix smart contracts. This iteration of the audit was focused on an updated version of the codebase and new scope. The report for the first iteration of the audit can be found here.

The purpose of this audit was to achieve the following.

- Ensure that the smart contracts functioned as intended.
- Identify potential security flaws.

Assessing the economics, game theory, or underlying business model of the platform were 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. There are a number of techniques that can help to achieve this, some of which are described below.

- Security should be integrated into the development lifecycle.
- Defensive programming should be employed to account for unforeseen circumstances.
- Current best practices should be followed when possible.

The second iteration of the audit had a modified scope and updated features. Notable changes since the first audit included:

- An inflationary SNX supply, which could be minted weekly.
- A new eternal storage pattern was used to store fee data.
- Rewards were issued for staking synths, which vested over a period of one year.
- Fees could not be withdrawn from the system if the correct collateralisation ratio was maintained, instead of enforcing a penalty on the fees.
- A new ERC20 compatible proxy pattern was used.
- Inverse priced synths, which allowed for shorting of currencies.
- Purgeable synths, which could be removed from the system as long as the total value of the synth was below a certain limit.
- The state of the fee pool was separated from the contract itself to facilitate upgrades.
- New front-running defenses in the form of oracle trading locks, gas limiting and a protection circuit (slashing).

At the conclusion of the audit one high risk issue, three low risk issues, and several informational level issues and design comments were open. The high risk issue pertained to the slashing defense mechanism used to protect against front-running, which could inadvertently result in the loss of legitimate user funds. The impact of the low risk issues included:

- Not minting the correct number of tokens proposed by the release schedule.
- Setting an exchange rate above the hard-coded maximum.
- Potential issues regarding the algorithm used to calculate the value of Synthetix Drawing Rights (XDRs).

The informational level findings included excessive owner and oracle privileges, a function susceptible to integer overflow, as well as minor deviations from best practice and suggestions that may improve the functionality or readability of the code.

In line with the previous audit, the code was once again found to be of a very high standard, as it was modularised, well-documented, defensive, and generally adhered to best practices.

The new proxy pattern that was implemented to permit third party apps to call ERC20 functionality through it was found to operate as intended.

It should be noted that some core components were explicitly omitted from the scope of the assessment, including:

- Escrow functionality.
- Oracle functionality.

The risk posed by the smart contracts can be further mitigated by using the following controls prior to releasing the contracts in a production environment.

- Perform an additional audit with the entire system in scope. Specifically, the oracle smart contracts.
- Use a public bug bounty program to crowdsource the process of identifying security vulnerabilities.
- Extend the test suite coverage.

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 Synthetix Smart Contracts

Project Name: Synthetix
Commits: fdd4782
Files: DelegateApprovals.sol, ExchangeGasPriceLimit.sol, EternalStorage.sol, ExchangeRates.sol, FeePool.sol, FeePoolEternalStorage.sol, FeePoolState.sol, Proxyable.sol, ProxyERC20.sol, PurgeableSynth.sol, RewardEscrow.sol, SupplySchedule.sol, Synthetix.sol

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, both manually and through the Truffle test suite provided. 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 generated by solidity-coverage of the Truffle tests included in the project is given below.

File	% Statements	% Branch	% Functions	% Lines
contracts/	86.96	72.77	80.62	86.42
DelegateApprovals.sol	100	100	100	100
Depot.sol	85.45	65	73.08	85.96
EscrowChecker.sol	0	100	0	0
EternalStorage.sol	14.29	100	13.64	9.52
ExchangeGasPriceLimit.sol	75	75	75	80
ExchangeRates.sol	100	96.15	100	99.05
ExternStateToken.sol	88.89	50	81.82	88.89
FeePool.sol	93.02	76.09	89.8	94.34
FeePoolEternalStorage.sol	0	0	50	0
FeePoolState.sol	100	70	100	100
LimitedSetup.sol	100	50	100	100
Owned.sol	100	100	100	100
Pausable.sol	85.71	66.67	100	87.5
Proxy.sol	62.5	25	66.67	58.33
ProxyERC20.sol	61.11	100	60	61.11

File	% Statements	% Branch	% Functions	% Lines
Proxyable.sol	83.33	75	85.71	86.67
PurgeableSynth.sol	100	75	100	100
ReentrancyPreventer.sol	100	50	100	100
RewardEscrow.sol	96.55	72.73	100	95.08
SafeDecimalMath.sol	100	100	100	100
SelfDestructible.sol	64.71	50	60	64.71
State.sol	100	100	100	100
SupplySchedule.sol	97.78	87.5	100	97.83
Synth.sol	98.59	56.25	100	98.63
Synthetix.sol	98.39	83.33	92.31	98.44
SynthetixEscrow.sol	27.08	22.73	25	25
SynthetixState.sol	37.5	10	84.62	39.39
TokenFallbackCaller.sol	100	100	100	100
TokenState.sol	100	100	100	100
contracts/interfaces/	100	100	0	100
contracts/test-helpers/	72.22	50	72.22	73.68
PublicSafeDecimalMath.sol	80	100	80	80
TokenExchanger.sol	62.5	50	62.5	66.67
All files	86.72	72.66	79.94	86.21

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 was conducted using Slither, Securify, as well as MythX. Tools such as the Remix IDE, compilation output, and linters were 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.

4.1 Synthetix Smart Contract

The Synthetix contract is described below.

ERC20 Token

The token should be ERC20 compliant with the following values.

Field	Value
Symbol	SNX
Name	Synthetix Network Token
Decimals	18

Locking Mechanism

A user should be able to lock SNX in the Synthetix system in order to issue a proportionate number of synths, up to the issuance ratio. The user should then earn fees from transactions of synths within the system, relative to their percentage stake in the system as a whole.

If the price of SNX increases such that the collateralisation ratio for a user goes below the minimum specified amount, a portion of their locked tokens should be unlocked up to the point where the collateralisation ratio reaches the minimum required collateralisation ratio again.

Alternatively, if the price of SNX drops, then the user's SNX should be locked within the contract and fee withdrawal disabled until either the price corrects itself or the user purchases enough SNX to match the required collateralisation ratio. In order to unlock the SNX, the user's collateralisation ratio must be below the issuance ratio. To allow fee withdrawals, the user's collateralisation ratio must be equal to or below `issuanceRatio * (1 + TARGET_THRESHOLD)`.

Fee Collection and Distribution

Fees collected through transfers and exchanges of synths should be tracked through a fee pool. The fee pool should track when SNX are locked within the system to provide collateral for the synths. Fees should be paid out proportionately after each period, which is within a defined fee period, to users depending on the amount of SNX locked and when they locked their tokens.

Rewards Schedule

It should be possible for the Synthetix system to mint SNX token rewards based on a predefined issuance supply schedule. Any user should be able to call the minting function when a supply of reward tokens becomes mintable and receive a minter reward for doing so. The SNX rewards minted should then be placed in a reward escrow system for one year after the claim date. Users should be able to withdraw their SNX rewards after the escrow period has elapsed. The rewards supply schedule should be as indicated in the following table.

Year	Increase	Total Supply
1	0	100,000,000
2	75,000,000	175,000,000

Year	Increase	Total Supply
3	37,500,000	212,500,000
4	18,750,000	231,250,000
5	9,375,000	240,625,000
6	4,687,500	245,312,500

Escrow Integration

It should be possible for the Synthetix system to integrate with an escrow system.

Note: This functionality was beyond the scope of this assessment.

Synth Integration

It should be possible for the owner of the Synthetix contract to add and remove synths to the Synthetix system.

Issue and Burn Synths

It should be possible for the Synthetix system to issue or burn synths.

Exchange Rates and Inverse Rates

It should be possible for the Synthetix system to integrate with an exchange rate contract, which should control the various exchange rates within the system. The rates should be set by an oracle contract. Rates should be considered stale if they have not been updated in more than the time specified in the contract. A holder of synths should be able to exchange their synths held in one currency key into another currency denominated Synth flavour for an exchange fee. It should also be possible to set inverse pricing rates for inverse equivalent currencies known as inverse synths. These inverse synths should allow users to short the price of their representative (non-inverted) currency. Inverse synths should also have oracle-defined liquidation price margins above and below the starting value.

Oracle Functionality

The oracle should maintain exchange rates within the system. It should also activate the protection circuit defense in the event that front-running transactions are detected.

Note: The oracle code was not available and the functionality was beyond the scope of this assessment.

Preferred Currency

*NOTE: This feature is now deprecated and no longer available

It should be possible for users to set a preferred currency, automatically converting any funds that they receive in synths to their preferred currency. If no preferred currency is set for a receiving address, the default currency should be the original currency. No fee should be charged on this exchange, as the user should be charged a fee for the transfer.

Synthetix Drawing Rights (XDRs)

The Synthetix Drawing Rights is a flavour of synths that is loosely based on Special Drawing Rights. Its exchange rate is based on a basket aggregate of currencies to minimise exposure towards any particular fiat currency. Fees are stored in this currency, and users can hold these synths if they want to lessen their exposure to any particular fiat currency.

Proxy and Upgrade Functionality

It should be possible to store the token balances in an external state contract to allow for seamless upgrades to the main functionality while preserving the state of the tokens and the Synthetix ecosystem.

Front-Running Protection

It should be possible to mitigate malicious front-running of the exchange rate updates by setting a gas price limit for each call to the exchange functionality. This functionality should be facilitated by an exchange gas price limit contract. The Synthetix contract should validate each exchange transaction using this gas price limit contract. It should be possible for the contract owner to adjust the maximum gas price of exchange transactions.

It should be possible for the oracle to disable exchanges during a price update through a price update lock.

Lastly, the oracle should be able to slash the funds of an exchange transaction when it detects extreme cases of front-running. This creates a disincentive for front-runners, making attacks against the system potentially more costly.

4.2 Synth Token Smart Contract

The functionality of the Synth token smart contract is described below.

ERC20 Token

The token should be ERC20 compliant with the following values.

Field	Value
Symbol	`_tokenSymbol` constructor parameter
Name	`_currencyKey` constructor parameter
Decimals	18

Proxy and Upgrade Functionality

It should be possible to store token balances in an external state contract to allow for seamless upgrades to the main functionality while preserving the state of the tokens and the Synthetix ecosystem.

Transfer Types

It should be possible to perform ERC20 transfers with tokens, however, a fee should be reserved on each transfer. It should be possible for either the sender or receiver to pay the fee. Additionally, the token should implement functionality similar to the ERC-223 standard, in that it should trigger a token fallback function in a smart contract that supports the functionality. However, unlike ERC-223, the transfer should not revert if the receiving contract does not support token fallback functionality.

5. Detailed Findings

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

5.1 High Risk

5.1.1 Dangerous Front-Running Defense Strategy

Synthetix.sol: Line 434-440

Description

The "Protection Circuit" defense mechanism used to protect against front-running was found to be an overly aggressive approach that could lead to loss of user funds. The mechanism operated by allowing the oracle to enable a switch when it detected front-running. When the switch was enabled, any call to the `exchange(...)` function would simply burn the tokens that were meant to be exchanged, slashing the front-runner's funds. The intention was for the oracle to send transactions with high gas prices to enable and disable the switch, only targeting the front-runner's transactions. However, it would be possible for legitimate users of the system to have their funds slashed due to network interruptions or latency delaying the oracle broadcasting the transaction to disable the protection circuit.

This slashing approach recently gained attention when a front-runner lost their funds due to it. More information can be found in the Synthetix blog post.

Remedial Action

Alternative front-running defense mechanisms have been implemented in ExchangeGasPriceLimit.sol in the form of SIP-7 and SIP-12.

SIP7: The oracle contract can disable calls to `exchange(...)` while exchange rates are being updated.

SIP12: The gas price for calls to `exchange(...)` must below a maximum threshold set by the oracle. If the price exceeds the limit, the transaction will revert. This mechanism is substantially less aggressive, as it is limited to specific transactions and is less sensitive to network conditions.

As such, it is recommended that the "Protection Circuit" strategy be deprecated from the codebase.

If the slashing approach is still deemed necessary, it is recommended that the functionality be adjusted to at least limit the scope of transactions affected by it. For example, when enabling the switch it would be possible to set a maximum gas price (set at a price that would make front-running infeasible) and only slash transactions above that threshold, protecting ordinary users of the system.

Response from Synthetix Team

We have migrated Forex price feeds to Chainlink (SIP-32) and soon the rest of the crypto prices will move to Chainlink (SIP-36). Using fee reclamation (SIP-37), described in more detail in Issue-293 will reclaim the risk free profits from front runners. At that stage the protection implementations outlined in SIP-6, SIP-7, and SIP-12 will be deprecated.

5.2 Medium Risk

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

5.3 Low Risk

5.3.1 Unclaimed Mint Tokens are Lost After One Year

SupplySchedule.sol: Line 213

Description

When calling `updateMintValues()` to establish the number of tokens that could be minted, the function `_remainingSupplyFromPreviousYear(...)` was used to calculate whether any tokens were unclaimed from the previous period. Line 218 would then deplete the previous period's mintable supply. This implementation required `updateMintValues()` to be called at least once in a period to accurately track the number of available tokens.

For example, if in period *n* `updateMintValues()` was called, but was not called at least once in period *n-1*, any unclaimed tokens in *n-2* would not be included in the `_remainingSupplyFromPreviousYear(...)` calculation and be unaccounted for.

Remedial Action

Given that no significant inconsistencies were introduced into the system (e.g. incorrectly tracking the total supply of the token) through this edge-case and that realistically, it is highly unlikely that `updateMintValues()` would not be called at least once within a period, no further action is necessarily required.

However, a simple solution would be to iterate through all previous periods and perform the same calculation as on line 195 to determine whether any previous

period had remaining tokens available.

Response from Synthetix Team

A new Inflationary Supply Schedule has since replaced this one in SIP-23 Inflation Smoothing.

5.3.2 Dangerous Method for Maintaining Synthetix Drawing Rights (XDRs) Rate

Description

The XDR exchange rate was found to spike when adding or removing currencies from the basket. This could result in inconsistencies within the system.

The XDR exchange rate was calculated as the summation of the exchange rates within the basket, e.g.

```
sUSD = 1, sEUR = 1.25, sAUD = 0.5
XDR = 1 + 1.25 + 0.5 = 2.75
```

While the XDR currencies were hard-coded in ExchangeRates.sol, the basket could be modified by deleting a rate or through a system upgrade.

Consider the following example - in an early version of the exchange rates smart contract, 1,000 XDRs are issued with an exchange rate of 2.75 (i.e 2,750 USD worth of XDRs in the system). Then an upgrade to the exchange rate contract simply adds sGBP to the basket with an exchange rate of 1.25, and no new tokens are minted. The resulting XDR exchange rate would be 4.0, thus the amount of XDR represented in the system would become worth 4,000 USD, arbitrarily increasing the value by 1,250 USD.

This example highlights the point that it should not be possible to at any stage change the basket given the current method for calculating the XDR exchange rate. This includes adding or removing a currency.

Remedial Action

It is recommended that the XDR algorithm be modified to use weighting factors to better accommodate changes in the XDR basket to avoid potential future functional or security issues. However, this would introduce additional complexity into the

system for a one-time operation that could pose a substantial risk to the integrity of the system.

Response from Synthetix Team

XDR's have been removed from the codebase to be replaced with just using sUSD as the system denominator which is always 1. in SIP-33 Deprecate XDR synth from Synthetix.

5.4 Informational

5.4.1 Excessive Owner Capabilities

General

The owners of the contracts in the system were found to hold a substantial amount of control over the system. This was a design decision that permits for maintenance, upgradeability, and administration of the system. While this decision might be necessary for a certain stage of the project, it does introduce risk in the form of single points of failure throughout the system. Examples of this include being able to arbitrarily update and control the oracle (e.g. effectively allowing the modification of balances of accounts holding synthetix and synths), and changing the mechanics of the system.

This functionality could be exploited by a rogue actor (e.g. a Synthetix team member with malicious intent) or by a compromised private key and is noted as it extends the potential attack surface of the system. Ideally, the extent of owner capabilities should be limited over time and eventually removed. While the Synthetix team could conspire against their users, it would undermine the integrity of the system and would act against their best interests in ensuring the stability and long-term viability of the network.

As a user of the Synthetix system, it is important to keep in mind that this increases the risk of real-world influence, as a centralised system is more likely to be affected by legislative or political events. Historical examples of this include IDEX and ShapeShift introducing Know Your Customer (KYC) requirements for their users.

The risk of a single rogue actor or a compromised private key can be mitigated by ensuring that a high standard of security is observed by the owner account. For example, the owner account should be a multi-signature wallet, signed by properly secured hardware backed addresses.

Response from Synthetix Team

As the system matures via regular releases of enhancements and new mechanisims the Synthetix team relinquish control to the community. In order to support the upgradeability of the contracts the owner requires these privileges, once the system is ready these capabilities will be removed and controled by the Synthetix Protocol DAO. Read more about the Transition to decentralised governance on our blog.

5.4.2 Oracle Risks

General

Similar to the owner role, the oracle role holds a necessary, substantial influence over the system. This includes operations such as setting gas limits (which if faulty could deny service to the system), slashing account funds through the "Protection Circuit" defense mechanism, and the obvious concern of interpreting off-chain data to track exchange prices.

The oracle code is currently closed source and was excluded from the scope of this audit. It is advised that the Synthetix team perform a comprehensive security audit of the oracle code, and releases a redacted version of the report that reassures users of the system's integrity without disclosing sensitive information.

Response from Synthetix Team

We have migrated Forex price feeds to Chainlink in SIP-32 and soon the rest of the crypto prices will move to Chainlink in SIP-36 and the synthetix oracle will be deprecated. Check our blog.synthetix.io for the release roadmap and release announcements.

5.4.3 Design Comments

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

Unnecessary Validation

The `_remainingSupplyFromPreviousYear(...)` function in SupplySchedule.sol was found to perform an unnecessary check for whether a `uint` was less than 0 on lines 198-200. It is recommended that this functionality be removed in order to improve gas performance. Additionally, no other validation is required as a safe `sub(...)` function is used to perform the calculation that sets the `amountInPeriod` variable.

Use of Implicit Integer Sizes in Contract Implementations

It was found that the implicit type `uint` was used consistently throughout the project. It is suggested that these `uint` type declarations be replaced with the more explicit `uint256` type declaration, where appropriate, in order to adhere to best practice.

Response from Synthetix Team

While the size of default uint could change between solidity versions, it would be a significant breaking change. At this stage we have quite a few of these around the code base, so rather than changing it, we believe the best approach is to leave it for the time being, changing it to a more specific type if required (e.g. if uint changes in a breaking solidity version).

Unaudited Compiler Version

Version 0.4.25 of the Solidity compiler was used throughout the project. At the time of writing, no publicly available security reviews of the target version were available. A comprehensive security review was performed by the Zeppelin team on version 0.4.24 with several issues being identified and rectified in 0.4.25. As a result, no change is recommended. However, it is worth noting that there is a possibility that new, unknown vulnerabilities exist in the version used.

Response from Synthetix Team

0.4.25 was prefered over 0.4.24 with the known issues being rectified.

Multiple Variables Shadowing State Variables

It was found that in several instances, variables were declared with definition names that matched and/or shadowed existing state variables. Although in this case not strictly a security vulnerability or functional issue, it is recommended that these local variables be renamed so as not to mistakenly overshadow any state definitions. Some cases of shadowing identified in the code base are outlined below.

- *Synthetix.sol: Line 328* - `availableCurrencyKeys` variable shadows function name.
- *ProxyERC20.sol: Line 73, 85* - `owner` variable shadows contract state variable.

Variables Not Initialised

It was found that multiple primitive-type variables were uninitialised. Although in this case no functional issues occurred as a result, it is recommended that all numerical-type variables be initialised to zero. These cases are outlined below.

- *FeePool.sol: Line 369, 491* - `rewardPaid` is not initialised.
- *FeePool.sol: Line 370, 451* - `feesPaid` is not initialised.

Use of Default State Variable Visibility

It was found that multiple mappings and one address variable were using implicitly defined visibility types. Although in this case not strictly a security vulnerability or functional issue, it is recommended that all variables and functions have explicit visibility types specified in order to maintain adherence to coding standards. These cases are outlined below.

- *EternalStorage.sol: Line 51 - 57* - It is recommended that the visibility be set to `internal`.
- *Proxyable.sol: Line 41* - It is recommended that the visibility be explicitly set.

5.5 Closed

5.5.1 No Maximum Exchange Rate Validation (Medium Risk)

FeePool.sol: Line 179

Description

The `setExchangeFeeRate(uint _exchangeFeeRate)` was found to not enforce the maximum exchange fee rate, despite the variable being declared and initialised, and the function comment describing that a limit was enforced. This would allow the owner to set an exchange rate above the maximum exchange rate limit.

Remedial Action

A `require` statement should be added to the function to ensure that the `_exchangeFeeRate` is below the `MAX_EXCHANGE_FEE_RATE` before setting the exchange rate.

Update

Addressed in e13024a.

5.5.2 Target Threshold Susceptible to Integer Overflow (Informational Risk)

FeePool.sol: Line 257-258

It was found that although the `setTargetThreshold(...)` function evaluated the minimum bound for the `_percent` parameter, there was no upper bound restriction applied. This combined with the lack of using the SafeMath library on the arithmetic operation used to set `TARGET_THRESHOLD` meant the value would be susceptible to an integer overflow should a large enough value be provided for the `_percent` parameter. This results in the validation on line 257 being bypassed entirely.

For instance, if in future the lower bound for `_percent` is changed to a value exceeding 0, the validation may be bypassed with an overflow attack allowing `TARGET_THRESHOLD` to equal any number. While not strictly a security vulnerability or functional issue, it is suggested that all instances that may result in an integer overflow be attended to in order to prevent future unintended behaviour. This risk can be mitigated by using the SafeMath library for the arithmetic operations used to calculate `TARGET_THRESHOLD`.

Update

Addressed in 5cc3c61.

5.5.3 Insufficient Unit Test Quality and Coverage (Informational Risk)

General

Certain areas of the codebase were found to have insufficient unit test quality and coverage. It is suggested that unit test code coverage be extended to cover additional functionality. Although no cases could be identified where the lack of coverage would hide a possible security vulnerability or functional issue, it is recommended that these cases be fixed and the test-base be extended to achieve optimal coverage of the cases presented below.

- *FeePoolEternalStorage.sol*: `importFeeWithdrawalData(...)` function untested.

Update

Test for `getPenaltyThresholdRatio()` added in 6a9f021.

5.5.4 Closed Design Comments (Informational Risk)

Misleading Function Implementation

The `removeInversePricing(...)` function in ExchangeRates.sol was found to emit an event regardless of whether a currency was actually removed. A more useful pattern would be to only emit an event if a currency is identified in the for loop.

It might also be helpful to return `true` after the event has been emitted, and then revert (or return false) at the end of the loop in the event that no valid currency is found. This would notify the caller to the fact that the function had been called with an invalid currency key.

Update

Addressed in fc730b4.

Incomplete Inverse Pricing Documentation

The inverse pricing mechanism in ExchangeRates.sol was found to be poorly documented within the comments in the codebase. Specifically, the `rateOrInverted(...)` function, which was fairly complex, was found to require additional information. This would include details such as the purpose of an inverse price, freezing behaviour, and perhaps an example of an inverse rate being adjusted inversely and ultimately being frozen, much like the example given here.

Update

Addressed in fc730b4.

Secure your system.

Request a service

START NOW →