Blaize.Security

March 16th, 2023 / V. 1.0

SYNTHEX
SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

Synthex contracts' source code was taken from the repository provided by the Synthex Protocol team.

SCORE **9.6**/10

The scope of the project includes Synthex set of contracts:

contracts\token
SyntheXToken.sol
EscrowedSYX.sol

contracts\token\redeem
Crowdsale.sol
BaseTokenRedeemer.sol

contracts\pool
PoolStorage.sol
Pool.sol

contracts\synthex
SyntheXStorage.sol
SyntheX.sol
AddressStorage.sol
AccessControlList.sol

contracts\synth
ERC20X.sol

Repository:

https://github.com/synthe-x/contracts

Branch: main

Initial commit:

- e793bf057d997ab30381c5d4361e8711877e52a5

Final commit:

- 350f868fc4be33b7678f41ea7f4224199c312b0d

TECHNICAL SUMMARY

During the audit, we examined the security of smart contracts for the Synthex protocol. Our task was to find and describe any security issues in the smart contracts of the platform. This report presents the findings of the Synthex smart contracts security audit of the **Synthex** smart contracts conducted between **March 16th, 2023**, and **April 6th, 2023.**

Testable code

The code is 100% testable, which corresponds to the industry standard of 95%.

The audit scope includes the unit test coverage, based on the smart contract code, documentation and requirements presented by the Synthex team. The coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies. However, to ensure the security of the contract, the Blaize.Security team suggests that the Synthex team launch a bug bounty program to encourage further active analysis of the smart contracts.

THE GRAPH OF VULNERABILITIES DISTRIBUTION:

The table below shows the number of the detected issues and their severity. A total of 34 problems were found. 33 issues were fixed or verified by the Synthex team.

	FOUND	FIXED/VERIFIED
Critical	3	3
High	2	2
Medium	3	3
Low	6	6
Lowest	20	19

SEVERITY DEFINITION

Critical

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

High

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

Medium

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

Low

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

Lowest

The system does not contain any issues critical to the secure work of the system, yet it is relevant for best practices.

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

We have scanned this smart contract for commonly known and more specific vulnerabilities:

- Unsafe type inference;
- Timestamp Dependence;
- Reentrancy;
- Implicit visibility level;
- Gas Limit and Loops;
- Transaction-Ordering Dependence;
- Unchecked external call - Unchecked math;
- DoS with Block Gas Limit;
- DoS with (unexpected) Throw;
- Byte array vulnerabilities;
- Malicious libraries;
- Style guide violation;
- ERC20 API violation;
- Uninitialized state/storage/ local variables;
- Compile version not fixed.

Procedure

We checked the contract for the following parameters:

- Whether the contract is secure;
- Whether the contract corresponds to the documentation;
- Whether the contract meets the best practices in the efficient use of gas, code readability.

Automated analysis:

We scanned the contracts using several publicly available automated analysis tools such as Mythril, Solhint, Slither, and Smartdec. All issues found were verified manually.

Manual audit:

We manually analyzed the smart contracts to identify potential security vulnerabilities. Our analysis involved a comparison of the smart contract logic with the description provided in the documentation.

EXECUTIVE SUMMARY

The audited protocol is a synthetics platform based on the overcollaterization principle. The platform receives collateral deposits from users and mints synthetics by the linear utilization rate together with debt tokens for the user. Therefore the platform also combines features from the lending protocol since it enables the liquidation process for accounts under the collateral requirement. For user incentivization, the platform provides rewards distributions for liquidity provides and SYX token providers (via the esSYX contract). More info about the protocol (including admins functions) is described in the "Protocol Overview" section.

Firstly, the Blaize auditors team must mention that protocol has its own Oracle contracts, which utilize Chainlink, Aave, and Compound feeds. Though, Oracle contracts themselves are out of the scope of the current audits. So, the Blaize Security team highly recommends utilizing several price sources, having a TWAP-like solution, or using Chainlink or other oracles aggregating several sources. Dependency of the synthetic on a single price source (like Compound or Aave) increases the risk of price manipulation through the 3rd party pool manipulation.

From other side, the security team performed an extensive manual audit during the testing stage and covered several checklists for common attack vectors and possible exploit possibilities. The team found several critical and high-risk issues with collateral withdrawal, fee calculation during the liquidation, and ETH transfer.

Also, another noticeable issue was connected to ETH and WETH's identical support. So, after the audit, the SyntheX team added WETH support with mandatory ETH wrapping in the protocol. Also, most protocol contracts, including the Synthetic contract itself, collateral pool, and rewards pool, are upgradeable. The SyntheX team successfully fixed or verified all issues, and the security team rechecked fixes with appropriate tests.

The code is well documented with natspec and internal comments and contains important examples in crucial code places. The project has good native coverage for main user scenarios. The overall security is high enough, though the security team decreased the final rating a bit because of the dependency on oracles, which were out of the scope, the centralization aspect because of contract upgradeability, and several issues where the team has taken responsibility at. Tests coverage mark shows the evaluation of the protocol's native coverage.

From all aspects, SyntheX protocol has passed the security audit.

	RATING
Security	9.5
Gas usage and logic optimization	9.7
Code quality	10
Test coverage	9.2
Total	9.6

PROTOCOL OVERVIEW

SyntheX is a decentralized finance (DeFi) system that allows users to mint synthetic assets backed by collateral assets. The main functionality of SyntheX are:

1. Reward Distribution: The distribution of rewards for liquidity providers in various pools.
2. Collateral Management: Management of collateral assets, including depositing, withdrawing, enabling/disabling collateral, setting collateral caps, and volatility ratios.
3. Trading Pools: The contracts enable/disable trading pools and manage volatility ratios.
4. Administration: The contracts include functions for managing access control, pausing/unpausing the contract, and updating addresses for various components.

SyntheX contract handles reward distribution and managing access control.

Pool manages collaterals and volatility ratios.

ERC20X contract includes functions for minting, burning, swapping, and liquidating. The contract also implements a flash mint function, allowing users to borrow tokens for a short period with a specified fee, payable to the SyntheX system's vault.

Crowdsale contract allows users to buy SYX tokens (SyntheX) with ETH or ERC20 tokens at a fixed rate during a specified timeframe.

SyntheXToken contract is an ERC20 token contract with additional features such as burnable, pausable, permit (gasless approval), and access control based on the SyntheX system contract.

EscrowedSYX contract is an ERC20 token contract with additional features such as staking, rewards distribution, and access control. The main features of this contract include the following:

1. Locking tokens: The lock function lets users lock their SYX tokens and receive escrowed SYX (esSYX) tokens in return.
2. Unlocking tokens: The startUnlock function enables users to start the process of unlocking their SYX tokens by burning their esSYX tokens.
3. Claiming unlocked tokens: The claimUnlocked function allows users to claim their unlocked SYX tokens.
4. Staking and rewards: Users can earn rewards using another token (specified by the REWARD_TOKEN address) by holding esSYX tokens. The rewards can be claimed using the getReward function.
5. Voting: The contract supports voting using the ERC20Votes extension.

The _transfer function is also overridden to enforce that only authorized senders can transfer esSYX tokens.
AccessControlList contract is for managing role-based access control using OpenZeppelin's AccessControlUpgradeable and Initializable contracts. It defines three roles: L0_ADMIN_ROLE, L1_ADMIN_ROLE, and L2_ADMIN_ROLE, each with different levels of authority.

SYNTHE-X

SynteX.sol

L2Admin

pause()

unpause()

setPoolSpeed()

Push specified reward token to specified pool, set speed (reward per sec) for this reward token

removeRewardToken()

Remove specified reward token from reward tokens list

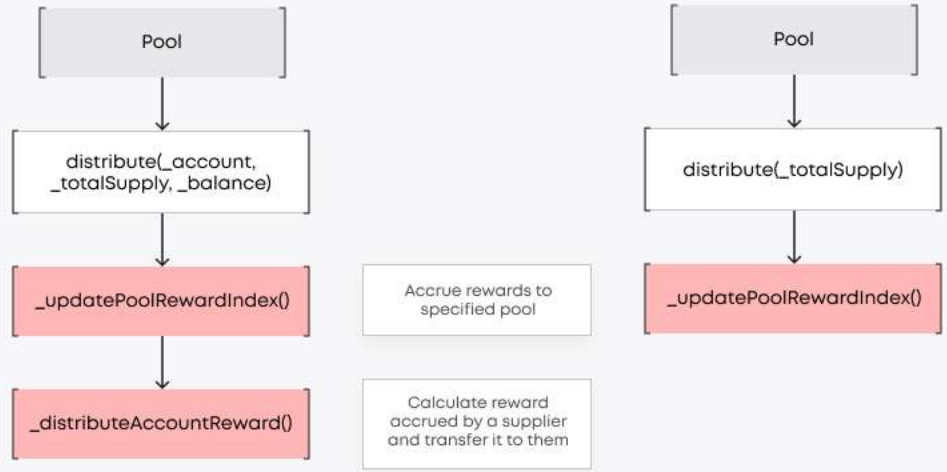
L1Admin

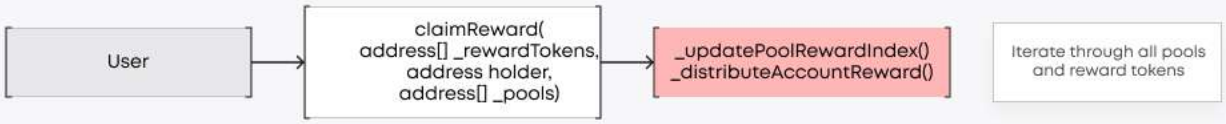
setAddress()

Owner

initialize()

setup L0, L1, L2 admins (used in other platform contracts)





SynteXToken.sol

L2Admin

pause()

unpause()

L1Admin

mint()

Owner

initialize()

need SyntheX address for initialization

S Y N T H E - X

Crowdsale.sol

L2Admin

pause()

unpause()

L2Admin

updateRate()

Set rate for a specific token, used when buying SYX with this token

endSale()

withdraw()

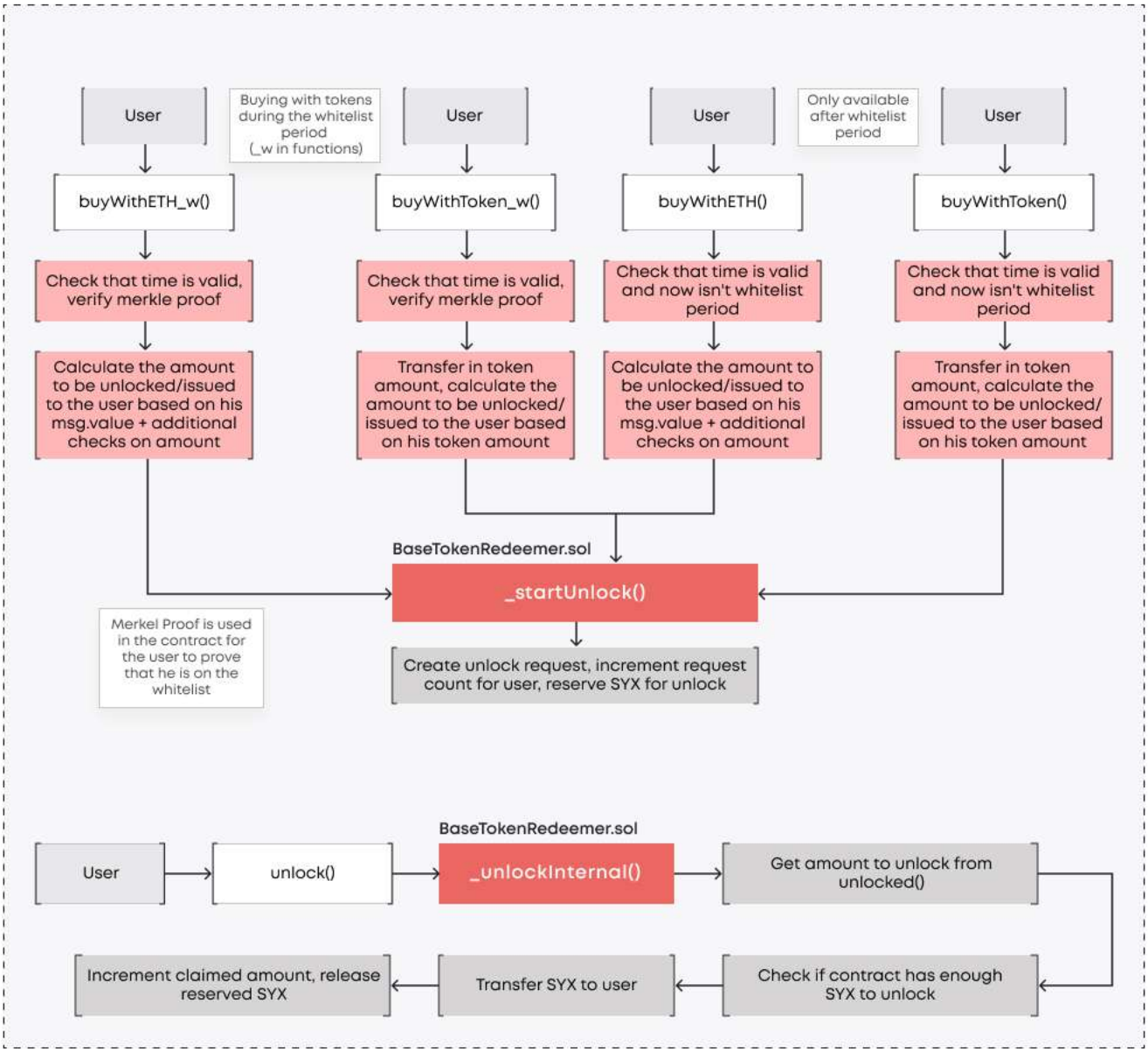
Withdraw ETH/ERC20 tokens from contract

Owner

initialize()

SyntheX address is needed. There are also arguments needed to limit the whitelist period (endTime, startTime and other)

The BaseTokenRedeemer contract is responsible for the token lock/unlock



S Y N T H E - X

EscrowedSYX.sol

L2Admin

pause()

unpause()

L1Admin

notifyReward()

setRewardsDuration()

grantRole()
revokeRole()

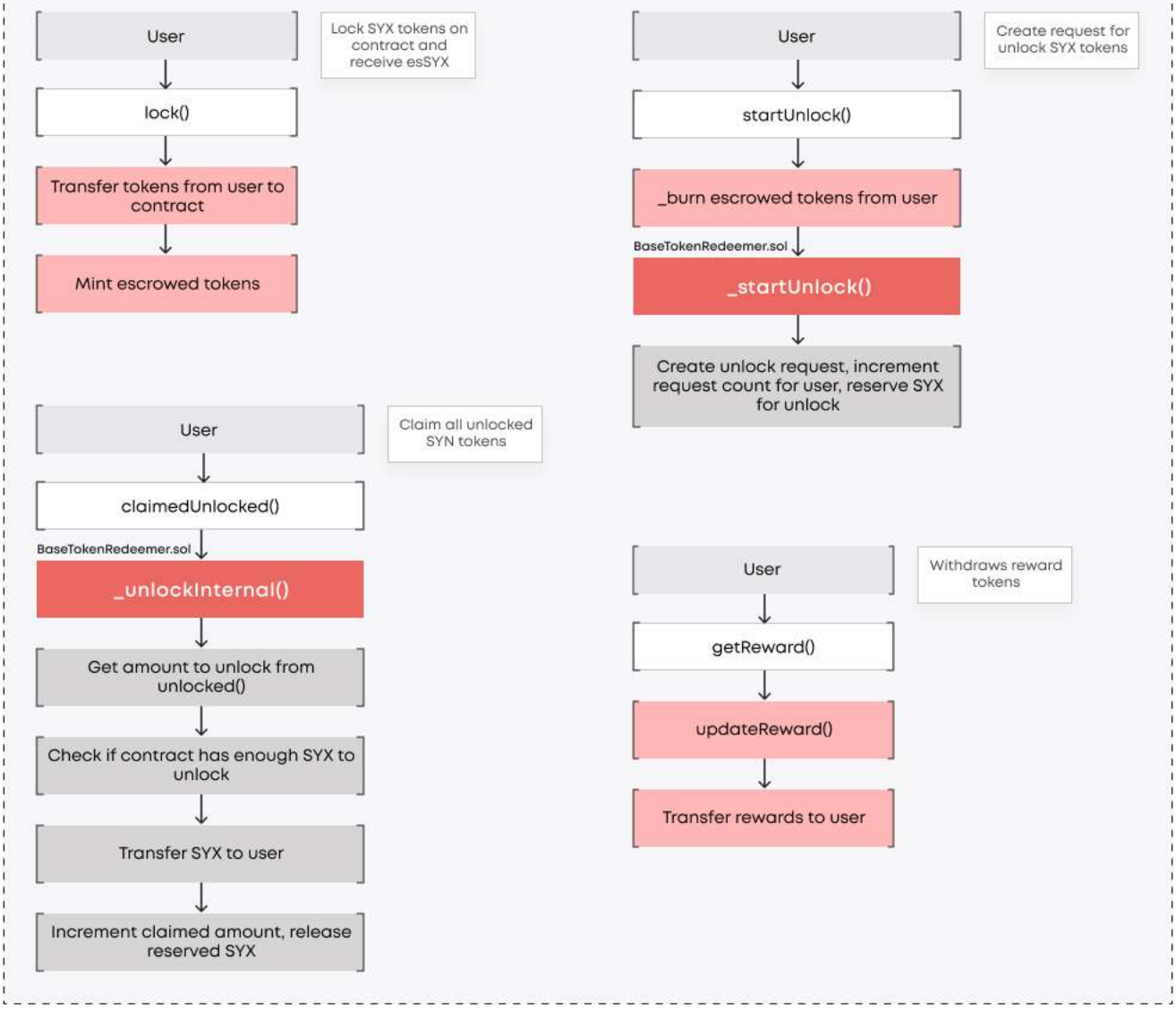
setLockPeriod()

Owner

initialize()

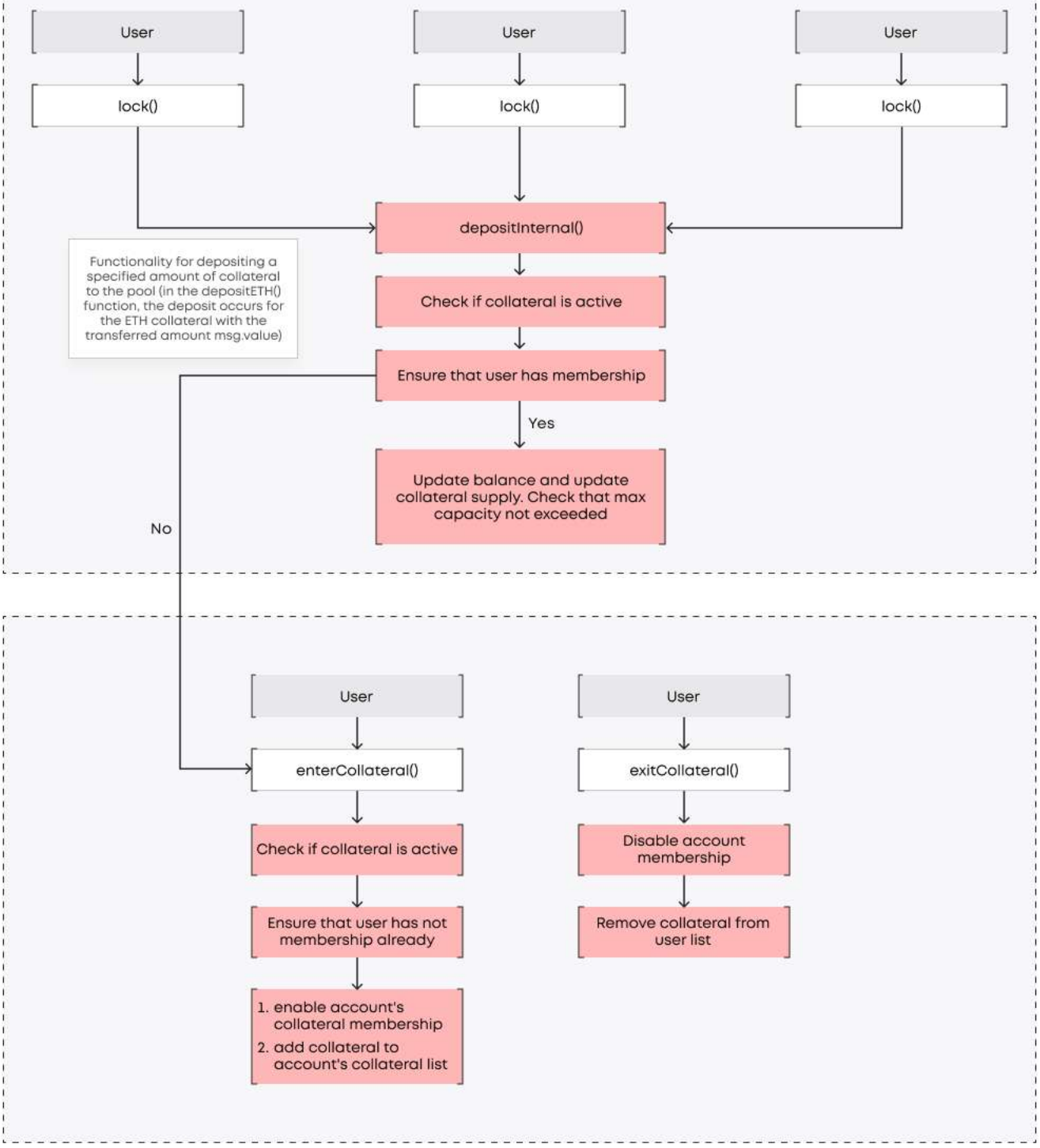
SyntheX address is needed. There are also arguments needed to limit the whitelist period (endTime, startTime and other)

The BaseTokenRedeemer contract is responsible for the token lock/unlock



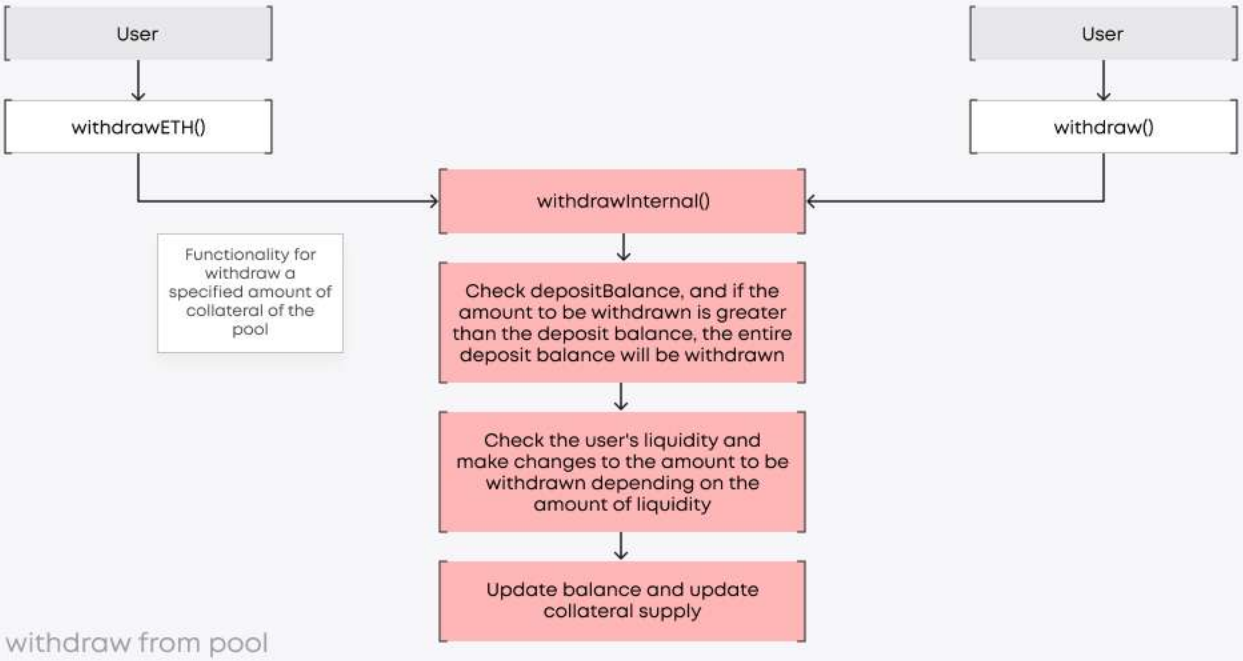
S Y N T H E - X

Pool.sol



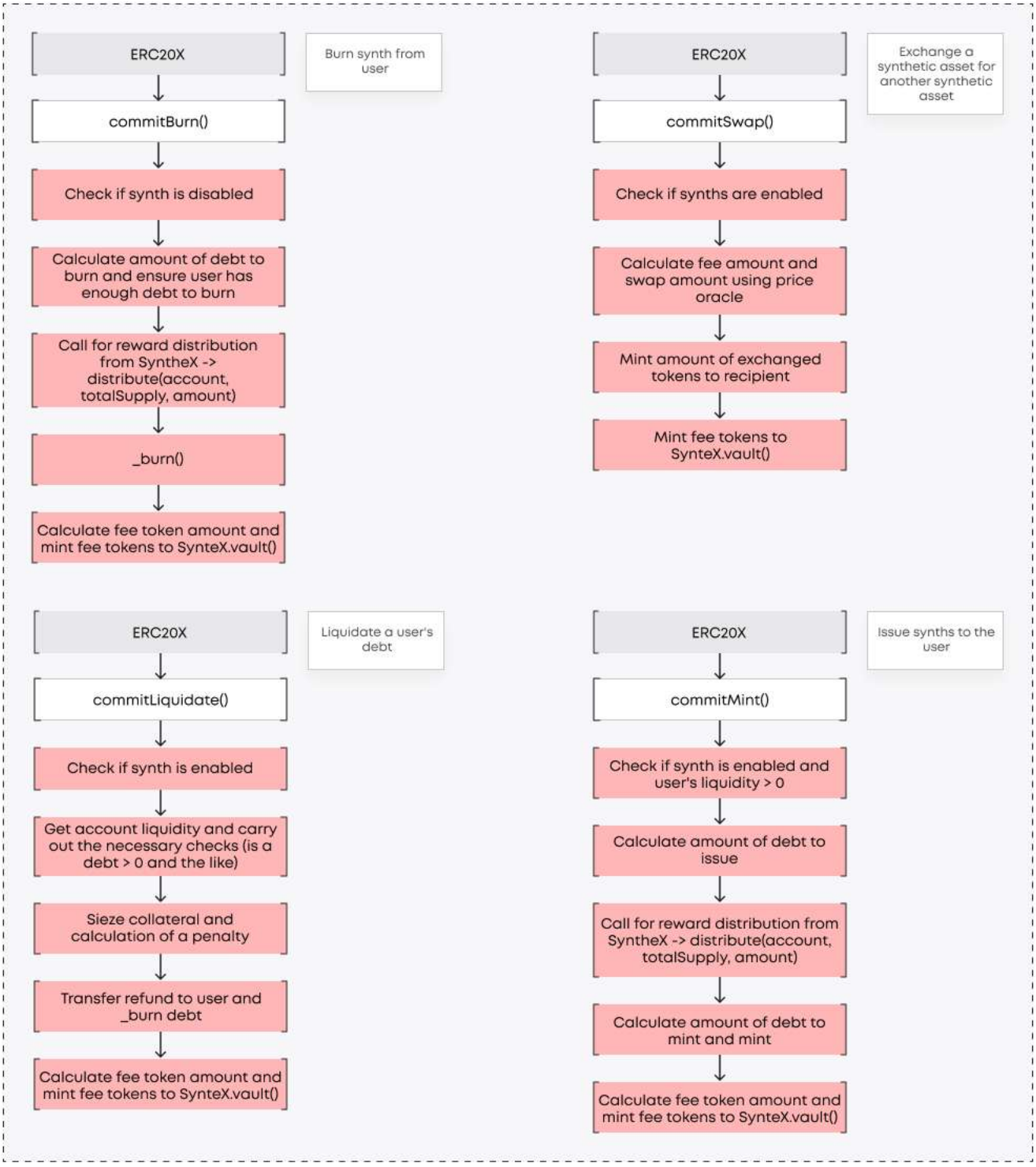
SYNTHE-X

Pool.sol



S Y N T H E - X

Pool.sol



S Y N T H E - X

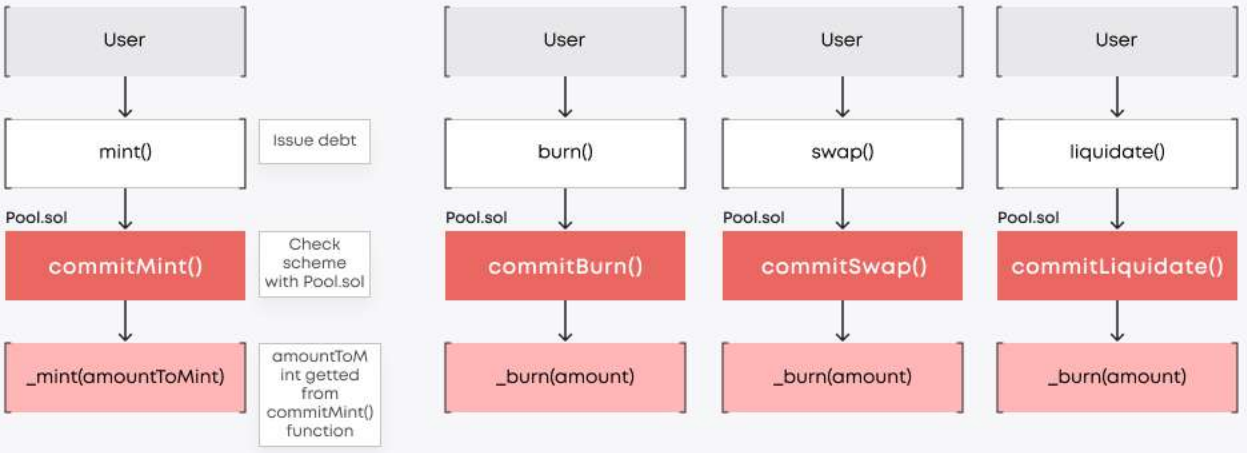
ERC20X.sol

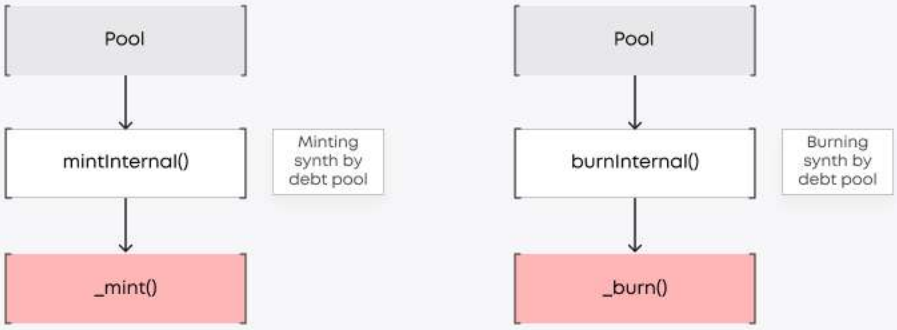
L1Admin
updateFlashFee()

Fee charged for flash loan

Owner
initialize()

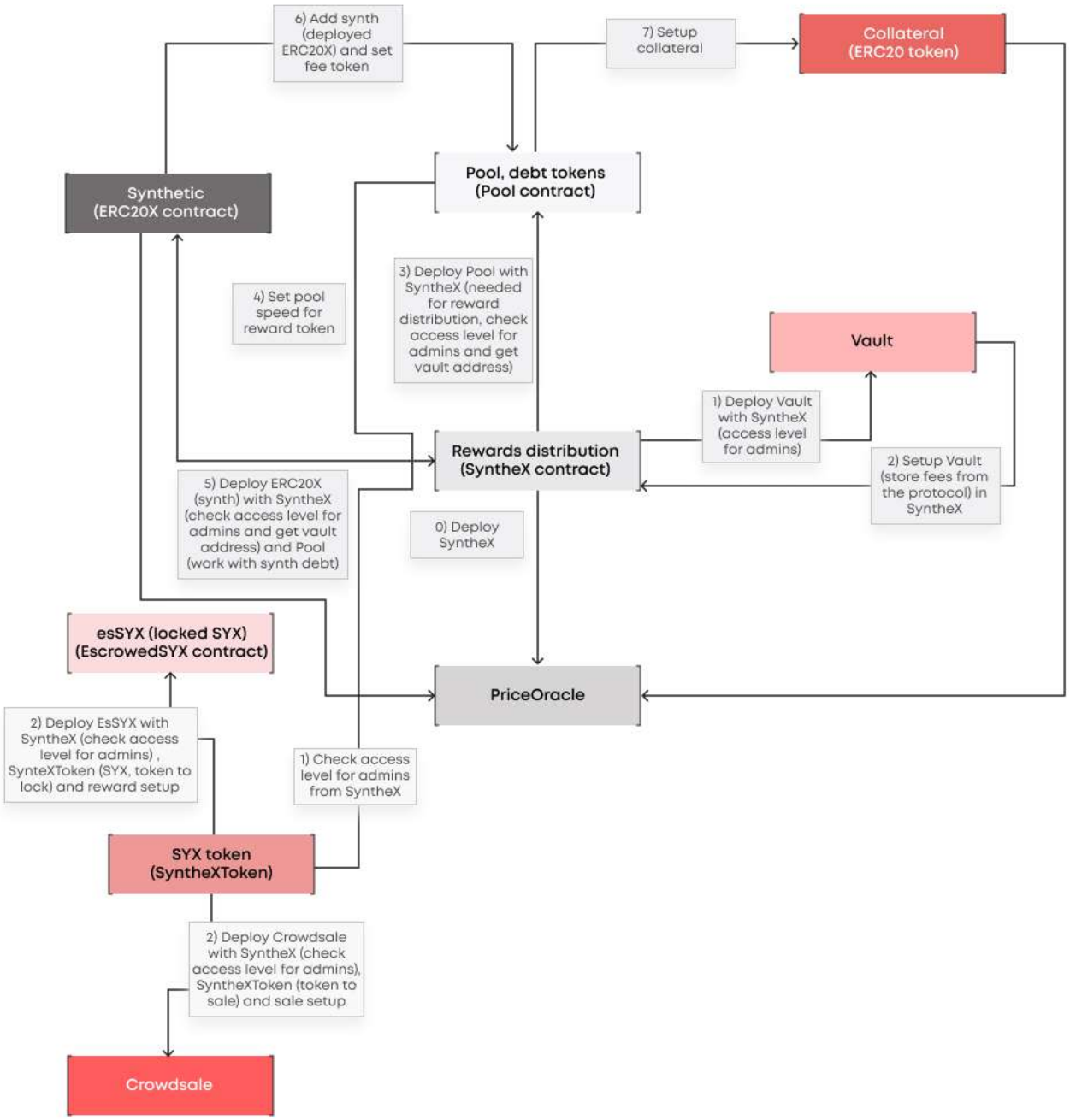
SyntheX and Pool addresses are needed.





S Y N T H E - X

Synthe-X protocol



COMPLETE ANALYSIS

CRITICAL-1 ✔ **Resolved**

User can withdraw extra funds

Pool.sol: All withdraw functions.
Scenario 1:

1. User1 deposits collateral
2. User2 withdraws collateral

Scenario 2 (for checks after the obvious fix)
1) The user deposits collateral1 and incurs debt to make getAccountLiquidity() negative (this can be achieved with price manipulation via flash loans).
2) The user withdraws any amount of collateral2. Due to the absence of checks and because the amount is set to 0 (due to negative liquidity) in withdrawInternal(), the contract successfully updates the storage.
3) User gets collateral2 as it is sent in the withdraw functions without checks.

As a result, the user can withdraw any available amount of any collateral from the contract. There are no restriction for the user: the contract withdraws and transfers funds first, then just accepts any accountCollateralBalance of the user (even 0) without checking if the user ever deposited the collateral or has enough balance for the withdrawal.
Also, others factors affected by this issue include: the deposit amount withdrawn does not equal the sent amount for the user; affected by this issue include (since the contract sends just everything left in case of nonsufficient balance on the contract).

Recommendation:

- Review withdraw logic
- add a check for the market entrance before the funds transfer.

- provide restrictions based on accountCollateralBalance and collateral availability - before the transfer
- add checks and reverts in case of insufficient amounts on the contract (both for tokens and ETH)
- revert if the account falls off the collateral criteria
- make funds transfer AFTER all storage changes - including recalculation of the amount based on the available deposit, and recalculation based on the liquidity available
- also, it is generally not recommended to decrease the withdrawn amount during the transaction without appropriate warnings in the dApp interface and documentation - and currently, these flows are not described
- the general recommendation is to revert the transaction in case of any failed check - instead of decreasing the withdrawn amount
- verify all calculations for user liquidity - for users without deposits
- uncomment and finalize health checks.

It is also highly recommended to conduct verbal checks against available collateral instead of relying on underflow reverts. This impacts the readability of the code and revert investigations.

CRITICAL-2 ✔ **Resolved**

Incorrect currencies conversion for the withdrawal.

Pool.sol, withdrawInternal()

_amount parameter of the withdraw function is in the collateral currency (ETH or any other token), while the liquidity variable (which holds the result of the getAccountLiquidity() call) is in USD. Though, the contract still assigns liquidity directly to _amount, resulting in a completely wrong amount.

Recommendation.

Review the withdrawInternal() logic and correct the currency conversions so that _amount, depositBalance, and totalDeposits are always in the same currency.

Post-audit.

The currency conversion was removed. Now, the function checks general collateral requirements.

CRITICAL-3	✔ Resolved

Incorrect ETH amount transferred.

Pool.sol: transferOut().
The function sends msg.value of ETH instead of _amount.
For now user cannot withdraw funds.

Recommendation:

Correct transferred ETH amount.

HIGH-1	✔ Resolved

ETH transfer may fail.

Pool.sol, transferETH()
The function utilizes the .transfer() method to transfer ETH. The transferETH() function sends ETH to an account, which may be set to a multisig account. In this case, transfer() may revert because it does not forward enough gas, causing funds to become stuck on the contract.
Since the .transfer() and .send() methods became obsolete after the Istanbul Ethereum update, it is recommended to use .call() for transfering funds. This issue is marked as high because the receiver address is provided by users, and a smart contract account could potentially be used.

Recommendation:

Use .call() for ETH transfer with the check of the .call() result.

HIGH-2	✔ Resolved

Fee is included twice during liquidation.

Pool.sol, commitLiquidate().
The liquidation process follows these steps:

1. Get _amount of synthetic1 from the liquidator.
2. Convert _amount into USD (line 427)
3. Increase the USD amount by the fee (USD * BASIS_POINTS/ (BASIS_POINTS - fee) (line 427). At this point, amountUSD covers both _amount and the fee.
4. Convert the USD amount into amountOut in synthetic2 (line 433).
5. Assume that the account debt covers amountUSD and the account collateral covers amountOut.
6. Move collateral from the account to the liquidator (line 467).
7. Convert amountOut back to USD (line 474) (so it still covers _amount and fee from step 3).
8. Burn account debt.
9. Mint fee.
10. Convert amountUSD back to the synthetic1 amount (line 492).
11. increase the synthetic1 amount by fee again (line 492): amountUSD.toToken() * (BASIS + fee) / BASIS -> it will increase amount again.

After all, excluding conversions and assuming that the amount to liquidate is covered by user's debt, we will have _amount from the liquidator increased twice by fee.
So, first of all, the liquidator covers fee twice; secondly, the function will revert in ERC20X.liquidate() since the returned amount will be greater than the initial one.

Recommendation:

Verify the fee covering process. It appears that the fee should be subtracted from the final amount on line 474 before burning the debt. Additionally, there should be no increase in the liquidator's amount during the return. The math needs to be verified from the Synthex team.

Post-audit:

Amount of synth1 to burn now includes burn fee just once and step (3) is corrected to exclude fee

MEDIUM-1	✔ Resolved

Checking ETH price instead of token price.

Crowdsale.sol, function buyWithToken, and buyWithToken_w.
The user can be assigned zero tokens (line 155) due to absence of check that token rate is not zero.

Recommendation:

Check that the price in ERC20 tokens is not zero.

MEDIUM-2	✔ Resolved

Possibly incorrect access control.

Pool.sol, updateCollateral(),
The NatsPec documentation says that governance or L2Admin can update the collateral. However, the function has strict restrictions for L1Admin. It appears that additional permission may be missing.

Recommendation:

Check the access control for pool admin functions and add L2admin if needed OR correct the NatsPec documentation.

Recommendation.
The NatsPec documentation is correct now: the function is restricted for L1 only.

MEDIUM-3	✔ Resolved

Pausable contract cannot be paused or unpaused .

ERC20X.sol is PausableUpgradeable but neither pause or unpause functions are implemented.

Recommendation:

Implement pause and unpause functions or remove the pausable feature from contract.

LOW-1	✔ Resolved

Inaccurate version pragma.

`pragma solidity ^0.8.0;` is used in all contracts. The contract should be deployed with the same compiler version and options that it has been most tested with. Locking the pragma version helps ensure the contract is not accidentally deployed using a different version. Additionally, older versions may contain bugs and vulnerabilities and be less optimized in terms of gas usage. Using the latest version of Solidity and specifying the exact pragma is recommended.

Recommendation:

Specify the latest version of Solidity in the pragma statement.

Post-audit.

Contracts use solc 0.8.19.

LOW-2	✔ Resolved

Missing validation.

ERC20X.sol: initialize() hasnt validation for pool.address and syntheX.address.
BaseTokenRedeemer.sol: initialize() hasnt validation for _TOKEN, _lockPeriod, _unlockPeriod, and _percUnlockAtRelease.
This allows these fields to be initialized with any address, not just the addresses of specific contracts, which can break the contract.

Recommendation:

Add validation pool.address and syntheX.address initializer.

LOW-3	✔ Resolved

Extra gas spendings on SafeMath.

SyntheX.sol, Pool.sol, ERC20X.sol, BaseTokenRedeemer.sol, EscrowedSYX.sol.
The contract utilizes SafeMath which became obsolete, since solc 0.8.x has built in support of overflow/underflow reverts. Consider removing the library usage for gas saving.

Recommendation:

Remove SafeMath usage.

LOW-4	✔ Verified

Confusing return values.

SyntheX.sol: getRewardsAccrued(), claimReward().
It is unclear what rewards should be returned for getRewardsAccrued() and what amounts transferred in claimRewards().
Consider scenario:

- pool1 with RewardTokenA
- pool2 with RewardTokenB
- user have deposits into both pools and mint ERCX tokens
- call getRewardsAccrued() for both pools and both reward tokens -> the contract will update rewardAccrued[user] for both reward tokens
- call get RewardsAccrued([RewardTokenA], user, [pool2]) -> the protocol will return rewardAccrued for RewardTokenA, though it is not bound to the pool2
- call claimReward([RewardTokenA], user, [pool2]) -> the protocol will transfer RewardTokenA, though the user calls function for the pool2

It happens because of the unclear flow for claiming: the contract has connection rewardToken -> user (in rewardAccrued mapping), rewardToken -> pool (in rewardState mapping) but functions for claim and accrue seem to use pool -> reward token connection. Therefore, either make claim and accrue functions working for all rewards independently from the pool (for both accrue and claim), OR use rewards connected to the pool (therefore recommendation connects with Info-11 about dedicated rewards.

This issue is marked as Low since there are no funds lost. But, we highly recommend working on the confusing logic (especially when users will provide additional deposits into several pools) since the impact on display and the process of how users receive tokens will become more confusing with the protocol growth.

Recommendation.

Remove the rewardTokens parameter from both functions and use dedicated rewards for the pool OR use functions for rewardTokens independently from the pool OR confirm current logic which will become confusing for several pools with several reward tokens.

Post-audit:

SyntheX team verified, that the dApp will resolve possible issues offchain, since onchain storage is not compromised.

LOW-5	✔ Resolved

Illogical storage structure.

Pool.sol: synthex, struct Vars_Burn, struct Vars_Mint

The pool contract utilizes a pattern for the storage separation from the implementation (into PoolStorage.sol). Though, the Synthex parameter remains in the Pool contract. That may cause future issues during further development or contracts upgrade (since the pool is an upgradeable contract).

The same applies to struct Vars_Burn and struct Vars_Mint definitions - it violates best practices and code readability, so it is better to move to the storage contract as well (to other structs definitions).

Recommendation:

Move the storage variable to the storage contract.

LOW-6	✔ Resolved

User cannot exit collateral while the deposited collateral balance is zero.

Pool.sol, function exitCollateral(), line 112. When a user has last collateral he cannot exit even if he has zero bad debt.

Recommendation:

Change the comparison from more than zero, to more or equal to zero in line 112.

LOWEST-1	✔ Verified

Custom errors should be used

Starting from the 0.8.4 version of Solidity it is recommended to use custom errors instead of storing error message strings in storage and use “require” statements. Using custom errors is more efficient in terms of gas spending and increases code readability.

Recommendation:

Use custom errors.

Post-audit.

The customer prefers numerated errors (in Compound style).

LOWEST-2	✔ Resolved

Duplicated code.

SyntheX.sol: line 5, 9.

The contract imports the UUPSUpgradeable.sol file twice.

Recommendation:

Remove one of the import statements to prevent confusion.

LOWEST-3	✔ Resolved

Redundant imports.

SyntheX: line 19.
The contract imports hardhat/console.sol`for debugging purposes, but it does not appear to be used in the code.
The same applies to EscrowedSYX.sol: draft-ERC20Permit.sol, ERC20.sol.
ERC20Votes inherit ERC20Permit, and ERC20Permit inherits from ERC20.
The import of SyntheXToken.sol: ERC20FlashMint is not used.
EscrowedSYX.sol: The import of "../synthex/SyntheX.sol" can be removed.
BaseTokenRedeemer.sol: The imports of SyntheXToken and SyntheX can be removed.
Crowdsale: SyntheXToken and SyntheX (after the fix according to info-12) can be removed.

Recommendation:

Remove redundant imports.

LOWEST-4	✔ Resolved

Duplicate `require`instead of modifier.

SyntheX.sol: pause(), unpause().
To improve the readability of the contract, you should use onlyL2Admin modifier existing in the contract instead of repeating the `require(isL2Admin(msg.sender),Errors.CALLER_NOT_L2_ADMIN)` in the functions listed above.

Recommendation:

Use a modifier instead of the duplicated `require` statement.

LOWEST-5	✔ Resolved

No error message in require.

Crowdsale.sol, line 98, 99, 110, 132, 146, 188.
Note: this issue is connected with the Info-1 - in case custom errors are used, apply custom errors for these cases as well.

Recommendation:

Add error message in require.

LOWEST-6	✔ Resolved

“Don't repeat yourself” code principle violation.

Crowdsale.sol, functions “buyWithETH_w”, “buyWithToken_w”, “buyWithETH” and “buyWithToken” have many lines of code in common and can be united in one function.

Recommendation:

In case of absence of critical need in having separate functions we suggest making a single function.

LOWEST-7	✔ Resolved

The presence of TODO in the code.

In contract ERC20X.sol lines 65, 77, 89, 100 have comments stating “TODO check if the amount is correct”. According to the logic of the contract, checking the correct amount is really required. The presence of "TODO" in the code is not recommended.

This issue is marked as Info. However, it may signal about missed or unfinished functionality and may be requalified.

Recommendation:

Add check to verify if the amount is correct. Remove the “TODO” comments.

Post-audit.

The contract now has the necessary validations.

LOWEST-8	✔ Resolved

Incorrect error.

ERC20X.sol, functions pause() and unpause(). Function check L2 admin but throw error for L1 admin.

Recommendation:

Use correct errors.

LOWEST-9	✔ Resolved

Role shouldn't be able to have such permissions by the context.

Pool.sol: initialize()

A pause() is called in the initialization (line 48) can only be called by L2Admin. Accordingly, deployment (and initialize() call) will be performed by L2Admin.

Recommendation:

Verify that L2Admin is responsible for the deployment OR call internal method _pause() for correctness.

LOWEST-10	✔ Resolved

Magic number used

SyntheX.sol: _distributeAccountReward().
The function uses 1e36 numeric constant (line 184) as a divisor for the accountDelta. Looks like it matches the rewardInitialIndex constant. If so - it is better to use that constant. If not - it is recommended to use a new constant.

Recommendation:

Use constant instead of “magic number” for better code consistency and numeric values matching.

LOWEST-11	✔ Resolved

Role-based restriction recommended.

SyntheX.sol: distribute() (both functions).
By default, both functions should be called by the Pool only (since they use reward tokens bound to msg.sender). Therefore, it is better to restrict the functions, as they are currently available to the public.

Recommendation:

Adjust the NatsPec documentation to reflect the accessibility of the functions.

LOWEST-12	✔ Resolved

Unoptimized import.

SyntheX.sol: setPoolSpeed() , claimReward(), getRewardsAccrued()
SyntheXToken.sol: SyntheX import.
ERC20X: SyntheX and Pool import.
The functions use the Pool contract to access the totalSupply() function. It is recommended to use the IPool interface instead. This requires adding the totalSupply() and balanceOf functions to the interface. Using IPool will decrease the contract size and eliminate the current cross-dependency between contracts.

The same applies to SyntheXToken.sol, Crowdsale, and ERC20X.

Recommendation:

Consider using the IPool interface (after its expanding for view functions) and ISyntheX.

Post-audit:

SyntheXToken, and Crowdsale now utilize the SyntheX interface, and SyntheX uses the Pool interface.

LOWEST-13	✔ Verified

Use dedicated reward tokens for pools.

SyntheX.sol: claimReward(), getRewardsAccrued().
The functions receive reward tokens as arguments, though passing any arbitrary tokens into it with a certain level of unpredicted behavior. Therefore, it is recommended to use a dedicated tokens store in rewardTokens mapping (since both functions also get a list of pools).
First of all - it will decrease the risk of unexpected behavior with arbitrary tokens.
Secondly - it will decrease the number of cycle leaps in internal functions.
This issue is marked as Info. Despite the risk of unexpected behavior with arbitrary tokens, exploit risk is very low with no positive scenarios for now. Though, since contracts are upgradeable and logic can be changed at any moment, the absence of arbitrary tokens sanitizing may cause a risk increase in the future.

Recommendation:

Remove _rewardTokens arguments from claimReward() and getRewardsAccrued() function and use tokens stored in rewardTokens mapping OR provide validation/sanitizing that tokens in the passed array correspond to the pools.

Recommendation:

SyntheX team verified, that possible display issues will be resolved offchain in the frontend part of the dApp.

LOWEST-14	✔ Verified

Referral is not used for any purposes

ERC20X: mint(), swap()
Both functions receive “referredBy” argument which is not used. There is no logic bound to the “referredBy”. This parameter is only emitted in the event. Since functions are public and not restricted, anyone can call them with any parameter passed as “referredBy”. So, it can influence the info shown in the dApp and historical data - and it may influence further development. This is also crucial, since there are no validations against the referral.

Recommendation:

Remove unused parameter OR provide validation OR move referrals to the part of the protocol where they are used.

Post-audit:

the team verified the necessity of “referredBy” parameters, and verifies that conditions for self-referring will be handled off-chain.

LOWEST-15	✔ Resolved

Missing fee validation

ERC20X.sol: updateFlashFee()
There is no validation against the BASIS_POINTS.

Recommendation:

Add validation for the fee to be not larger than BASIS_POINT.

LOWEST-16	✔ Resolved

ETH and WETH are different collaterals

Pool.sol: deposit functions
ETH has own descriptor for the collateral address (0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE) and from this point of view, there is no case to work with wrapped ETH (WETH), so it will be treated as different collateral.
This issue is marked as Info, since it refers to business logic rather than security. Therefore, it needs verification from the team. Though it also refers to an important use case, it should be mentioned in the report.

Recommendation:

Verify that WETH and ETH are different collaterals from the protocol perspective OR that WETH will not be used in the system.

Post-audit:

The team added direct WETH support. The contract stores WETH address and wraps ETH into WETH during the deposit, and users can choose unwrapping during the withdrawal.

LOWEST-17	✔ Verified

More debt than synthetics created on the mint

Pool.sol: commitMint()
The function provides the next calculations:

- gets requested amount of synthetic from ERC20X contract
- uses oracle to get USD equivalent of synthetic amount
- adds USD equivalent of minter fee
- mints debt tokens for the whole amount of USD equivalent + USD fee equivalent. Now the user has debt of USD amount + USD fee amount
- provides opposite conversion of USD equivalent into the synthetics amount (the function returns this number and ERC20X mints this amount of synthetics for the user)

- provides opposite conversion of USD equivalent of minter fee
- decreases minter fee by issuerAlloc percent
- mints the leftover (minterFee * (1 - issuerAlloc)) to the vault

Therefore we have (USD equivalent + USD minter fee equivalent) of debt tokens minted to the user and (synth amount + minterFee * (1 - issuerAlloc)) of synthetics minted. So minterFee * issuerAlloc of synthetics is never minted. Therefore, the system will have more debt than synthetics minted.
This issue is marked as **Info** because it is currently unknown whether it is a bug in calculations or desired logic. Though, it may be requalified as **High** after receiving comments from the team, as this approach may lead users to lose money due to excessive debt.

Recommendation:
Verify and correct the calculations so that the issuerAlloc of debt will actually be burnt.

Post-audit:
The team added a self-explaining example as a comment to the code.

LOWEST-18	✔ Verified

Confusing `require` statement.
ERC20X.sol: function swap(), line 96.
An amount is passed to the swap() ERC20X then passed to pool's commitSwap() function. Amount will not change in any way during all actions in the commitSwap() function. Then it turns out that on line 96 `require` checks whether the parameter amount is equal to the parameter amount.
Recommendation.
Remove the ` require`, **OR** validate the logic, perhaps the return value of commitSwap is not correct.
Post-audit.
Verified by the team to be necessary for future updates

LOWEST-19	Unresolved

Conflicting checks.

Pool.sol: commitSwap().

The function contains a requirement for synth to be active. Though, it also contains check for the inactive synth - but the function will revert in case of an inactive synth anyway.

Recommendation.

Verify the logic and correct the checks for inactive or disabled synths.

LOWEST-19	✔ Resolved

Liquidation bonus math

Pool.sol: commitLiquidate().

Looks like comments conflict with the code

The comment before the refund calculation states, that refund should be calculated if there is no enough collateral for the full penalty. Though, the refund itself is calculated in the opposite case. Partial penalty is paid if ltv*liqBonus > 1, but refund is calculated in the “else” statement.

Recommendation.

Verify the logic and either correct the comment OR correct the calculation.

Post-audit.

Bonus calculation is correct, the team updated comments in the code

contracts\token

SyntheXToken.sol
EscrowedSYX.so

✔ Re-entrancy	Pass
✔ Access Management Hierarchy	Pass
✔ Arithmetic Over/Under Flows	Pass
✔ Delegatecall Unexpected Ether	Pass
✔ Default Public Visibility	Pass
✔ Hidden Malicious Code	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass
✔ External Contract Referencing	Pass
✔ Short Address/Parameter Attack	Pass
✔ Unchecked CALL Return Values	Pass
✔ Race Conditions/Front Running	Pass
✔ General Denial Of Service (DOS)	Pass
✔ Uninitialized Storage Pointers	Pass
✔ Floating Points and Precision	Pass
✔ Tx.Origin Authentication	Pass
✔ Signatures Replay	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass

contracts\token\redeem

Crowdsale.sol
BaseTokenRedeemer.sol

✔ Re-entrancy	Pass
✔ Access Management Hierarchy	Pass
✔ Arithmetic Over/Under Flows	Pass
✔ Delegatecall Unexpected Ether	Pass
✔ Default Public Visibility	Pass
✔ Hidden Malicious Code	Pass
✔ Entropy Illusion (Lack of Randomness)	Pass
✔ External Contract Referencing	Pass
✔ Short Address/Parameter Attack	Pass
✔ Unchecked CALL Return Values	Pass
✔ Race Conditions/Front Running	Pass
✔ General Denial Of Service (DOS)	Pass
✔ Uninitialized Storage Pointers	Pass
✔ Floating Points and Precision	Pass
✔ Tx.Origin Authentication	Pass
✔ Signatures Replay	Pass
✔ Pool Asset Security (backdoors in the underlying ERC-20)	Pass

		contracts\synthex SyntheXStorage.sol SyntheX.sol AddressStorage.sol AccessControlList.sol
✓	Re-entrancy	Pass
✓	Access Management Hierarchy	Pass
✓	Arithmetic Over/Under Flows	Pass
✓	Delegatecall Unexpected Ether	Pass
✓	Default Public Visibility	Pass
✓	Hidden Malicious Code	Pass
✓	Entropy Illusion (Lack of Randomness)	Pass
✓	External Contract Referencing	Pass
✓	Short Address/Parameter Attack	Pass
✓	Unchecked CALL Return Values	Pass
✓	Race Conditions/Front Running	Pass
✓	General Denial Of Service (DOS)	Pass
✓	Uninitialized Storage Pointers	Pass
✓	Floating Points and Precision	Pass
✓	Tx.Origin Authentication	Pass
✓	Signatures Replay	Pass
✓	Pool Asset Security (backdoors in the underlying ERC-20)	Pass

		contracts\pool PoolStorage.sol Pool.sol
✓	Re-entrancy	Pass
✓	Access Management Hierarchy	Pass
✓	Arithmetic Over/Under Flows	Pass
✓	Delegatecall Unexpected Ether	Pass
✓	Default Public Visibility	Pass
✓	Hidden Malicious Code	Pass
✓	Entropy Illusion (Lack of Randomness)	Pass
✓	External Contract Referencing	Pass
✓	Short Address/Parameter Attack	Pass
✓	Unchecked CALL Return Values	Pass
✓	Race Conditions/Front Running	Pass
✓	General Denial Of Service (DOS)	Pass
✓	Uninitialized Storage Pointers	Pass
✓	Floating Points and Precision	Pass
✓	Tx.Origin Authentication	Pass
✓	Signatures Replay	Pass
✓	Pool Asset Security (backdoors in the underlying ERC-20)	Pass

		contracts\synth ERC20X.sol
✔	Re-entrancy	Pass
✔	Access Management Hierarchy	Pass
✔	Arithmetic Over/Under Flows	Pass
✔	Delegatecall Unexpected Ether	Pass
✔	Default Public Visibility	Pass
✔	Hidden Malicious Code	Pass
✔	Entropy Illusion (Lack of Randomness)	Pass
✔	External Contract Referencing	Pass
✔	Short Address/Parameter Attack	Pass
✔	Unchecked CALL Return Values	Pass
✔	Race Conditions/Front Running	Pass
✔	General Denial Of Service (DOS)	Pass
✔	Uninitialized Storage Pointers	Pass
✔	Floating Points and Precision	Pass
✔	Tx.Origin Authentication	Pass
✔	Signatures Replay	Pass
✔	Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY SYNTHEX TEAM

You are using the ùnsafeAllow.delegatecall`flag.
- ✓ supply token
- ✓ supply token with permit (48ms)

Rewards
- ✓ deposit with depositETH
- ✓ deposit by sending eth (86ms

Testing BurnFee

Burn fee
- ✓ should update fee to 1%
- ✓ user should be able to burn 1 sETH with 10 sUSD fee (119ms)
- ✓ should update fee to 0.1%
- ✓ user2 should issue synths (113ms)

Burned fee from isserAlloc
- ✓ should update fee to 1% + 50% issuer alloc
- ✓ user should be able to swap 10 sETH to 10000 sUSD with 50 sUSD fee + 50 sUSD burned (197ms)
- ✓ should update fee to 0.1% + 80% issuer alloc
- ✓ user should be able to burn 1 sETH for $1000 with 2 sUSD fee + 8 sUSD burned (229ms)

Testing MintFee

Minting fee
- ✓ should update fee to 1%
- ✓ user should be able to mints synths (70ms)
- ✓ should update fee to 0.1%
- ✓ user2 should issue synths (117ms)
- ✓ user2 should swap 1 seth to sbt

Burned fee from isserAlloc
- ✓ should update fee to 1% + 50% issuer allo
- ✓ user should mints synths (69ms)
- ✓ should update fee to 0.1% + 80% issuer alloc
- ✓ should user2 issue synths (119ms)

Testing SwapFee

Swap fee
- ✓ should update fee to 1%

- ✓ user should be able to swap 10 sETH to 10000 sUSD with 100 sUSD fee
- ✓ should update fee to 0.1%
- ✓ user2 should issue synth

Burned fee from isserAlloc

- ✓ should update fee to 1% + 50% issuer alloc
- ✓ user should be able to swap 10 sETH to 10000 sUSD with 50 sUSD fee + 50 sUSD burned (155ms)
- ✓ should update fee to 0.1% + 80% issuer allo
- ✓ user should be able to swap 10 sETH to 10000 sUSD with 50 sUSD fee + 50 sUSD burned (161ms)

Testing the complete flow

- ✓ Should stake eth (56ms)
- ✓ issue synths (176ms)
- ✓ swap em (93ms)
- ✓ update debt for users (126ms)
- ✓ burn synths (316ms)

Testing liquidation

Liquidation @ 85

- ✓ should not be able to liquidate if health factor is above 85% (129ms)

Liquidation @ 90

- ✓ user2 liquidates user1 with 1 BTC ($15000) (144ms)
- ✓ user2 completely liquidates user1 (125ms)
- ✓ tries to liquidate again (51ms)

Liquidation @ 99.5

- ✓ user2 liquidates user1 with 1 BTC ($15000) (279ms)
- ✓ user2 completely liquidates user1 (119ms)
- ✓ expect dusted account

Liquidation @ 100.5

- ✓ user2 liquidates user1 with 1 BTC ($20000) (139ms)
- ✓ user2 completely liquidates user1 (116ms)

Rewards

- ✓ set pool speed
- ✓ Should deposit eth (38ms)
- ✓ user1 and user2 issue debt (172ms)
- ✓ burn after 33 days (209ms)

- ✓ check esSYN rewards
- ✓ claim rewards (62ms)
- ✓ user2 burn remaining debt after 10 days (74ms)
- ✓ check esSYN rewards

Testing unlocker

- ✓ unlock should fail if esSYX balance is 0
- ✓ user1 gets 1000 esSYX
- ✓ user1 should be able to start unlock of 100 tokens
- ✓ user2 will start unlock of 250 tokens
- ✓ index0: unlock after lockPeriod, 0th of unlockPeriod, expect 5% to unlock
- ✓ index1: should not able to unlock
- ✓ index0: unlock after lockPeriod, 60/180 of unlockPeriod
- ✓ index1: unlock after lockPeriod, 53/180th of unlockPeriod

Testing Staking Rewards

- ✓ add 10 WETH reward for 1 year
- ✓ user1 should have 1000 esSYX
- ✓ view reward APY should be 100%
- ✓ user2 should stake 500 syn
- ✓ view reward APY 66%

Testing seal of esSYX

- ✓ lock syx
- ✓ should not be able to transfer
- ✓ should be able to transfer & transferFrom after getting authorized (53ms)

67 passing (9s)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
SyntheXToken.sol	42.86	25	60
EscrowedSYX.sol	65.71	39.58	69.57
Crowdsale.sol	0	0	0
BaseTokenRedeemer.sol	95.24	57.14	100
SyntheXStorage.sol	100	100	100
SyntheX.sol	81.82	47.47	64.29
AddressStorage.sol	100	100	100
AccessControlList.sol	90	50	85.71
PoolStorage.sol	100	100	100
Pool.sol	80.14	48.53	78.79
ERC20X.sol	76.19	42.31	63.64

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

Crowdsale

constructor

- ✓ start time cannot be in in the past (128ms)
- ✓ start time cannot be greater than end time (129ms)
- ✓ syntex cannot be zero address (125ms)
- ✓ token cannot be zero address (132ms)

whitelist buy with ether

- ✓ whitelisted user can buy tokens with ether (193ms)
- ✓ whitelisted user cannot unlock the same id twice (233ms)
- ✓ whitelisted user cannot buy tokens with before crowdsale start (60ms)
- ✓ whitelisted user cannot buy tokens with invalid proof (84ms)
- ✓ whitelisted user cannot buy tokens with for 0 ether (73ms)
- ✓ whitelisted user cannot buy more tokens than whitelist capitalization (91ms)
- ✓ user canot buy tokens when contract on pause (72ms)
- ✓ unlock cannot be called while contract on pause (65ms)

whitelist buy with token

- ✓ whitelisted user can buy tokens with payments tokens (186ms)
- ✓ whitelisted user cannot buy tokens with paymetn tokens before crowdsale start (53ms)
- ✓ whitelisted user cannot buy tokens with invalid proof (70ms)
- ✓ whitelisted user cannot buy tokens for 0 payment tokens (95ms)
- ✓ whitelisted user cannot buy more tokens than whitelist capitalization (153ms)
- ✓ user canot buy tokens when contract on pause (61ms)

buy with eth

- ✓ user can buy tokens with ether (226ms)
- ✓ user cannot buy tokens with ether before whitelist period ends (69ms)
- ✓ whitelisted user cannot buy tokens with for 0 ether (61ms)
- ✓ user canot buy tokens when contract on pause (90ms)

buy with token

- ✓ user can buy tokens with payment tokens (149ms)
- ✓ user cannot buy tokens with payment tokens before crowdsale start (39ms)

- ✓ whitelisted user cannot buy tokens for 0 payment tokens (59ms)
- ✓ user canot buy tokens when contract on pause (58ms)

admin function

- ✓ endSale
- ✓ admin cannot endSale after sale end (46ms)
- ✓ l1 admin can withdraw tokens from contract (73ms)
- ✓ l1 admin can withdraw ether from contract (121ms)
- ✓ l2 admin can unpause contract (92ms)
- ✓ recive works as buyWithEth function (326ms)
- ✓ fallback

ERC20X

- ✓ Should not mint if contract paused (44ms)
- ✓ Should not .mintInternal() if sender is not Pool contract
- ✓ Should not .burnInternal() if sender is not Pool contract
- ✓ Should not mint amount = 0
- ✓ Should not burn amount = 0
- ✓ Should not swap amount = 0
- ✓ Should not liquidate amount = 0
- ✓ Should update flash fee
- ✓ Should not update flash fee if sender is not L1 admin
- ✓ .burnInternal() should work correct (270ms)
- ✓ Should get flash fee from contract-inheritor (209ms)
- ✓ Should get flash fee receiver from contract-inheritor (190ms)

EscrowedSYX

Initialization

- ✓ Should initialize correctly (88ms)

Setters

- ✓ Should set rewards duration (49ms)
- ✓ Shouldn't set rewards duration if rewards period isn't completed (127ms)
- ✓ Should set rewards duration only by L2Admi
- ✓ Should set lock period (67ms)
- ✓ Should set lock period only by L2Admin (49ms)

Roles

- ✓ Should grant and revoke AUTHORIZED_SENDER role (112ms)
- ✓ Should revert when trying to grant or revoke roles without being L1Admin (134ms)

Pause/unpause

- ✓ Should pause (50ms)
- ✓ Should unpause (98ms)
- ✓ Should pause only by L2Admin (63ms)
- ✓ Should unpause only by L2Admin (78ms)
- ✓ Shouldn't call specified functions when paused (111ms)

Rewards operations

- ✓ Should notify reward (104ms)
- ✓ Should notify reward during rewards period (166ms)
- ✓ Should get amount of reward for duration (75ms)
- ✓ Should get reward per token (129ms
- ✓ Should get reward (157ms)
- ✓ Shouldn't get reward if notifyReward was't called (105ms)

Syx operations

- ✓ Should lock (92ms)
- ✓ Should start unlock (122ms)
- ✓ Should claim unlocked with 3 requests (262ms)

Transfer operations

- ✓ Should transfer esSYX only by authorized senders (147ms)
- ✓ Shouldn't transfer esSYX if sender is not authorized (114ms)

Pool

enterCollateral

- ✓ user can enter collateral (69ms)
- ✓ user cannot enter collateral twice (69ms)
- ✓ cannot enter not acitve collateral
- ✓ user can exit collateral with deposited collateral (128ms)

deposit

- ✓ user can deposit ERC20 (116ms)
- ✓ user can deposit ETH (73ms)
- ✓ user cannot deposit while contract on pause (60ms)
- ✓ user can deposit collateral he hasn't entered (105ms)
- ✓ user cannot deposit when collateral has exceeded capacity (107ms)

withdraw

- ✓ user can withdraw collateral (155ms)
- ✓ user can withdraw ETH collateral (133ms)
- ✓ user can withdraw ETH collateral as WETH (119ms)
- ✓ user cannot withdraw collateral he doesn't own (125ms)

mint
- ✓ user can mint (206ms)
- ✓ cannot mint with insufficient user collateral (294ms)

getAccountLiquidity
- ✓ getAccountLiquidity (435ms)

add/remove synth
- ✓ add synth (55ms)
- ✓ add 2 synths (76ms)
- ✓ cannot add same synth twice (84ms)
- ✓ only L1 admin can add synth
- ✓ remove synth number 1 out of 1 (84ms)
- ✓ remove synth number 2 out of 2 (99ms)
- ✓ only L1 admin can remove synth (75ms)

commitSwap
- ✓ commitSwap one synth for anoter synth (507ms)
- ✓ cannot commitswap disabled synth (45ms)
- ✓ only synth can call commitswap

commitLiquidate
- ✓ commitLiquidate (438ms)

PoolScenarios

Deposit/borrow/withdraw and reward calculation
- ✓ Should get rewards after repay if rewards off before repay (440ms)
- ✓ Should get rewards if rewards on after deposit and before repay (527ms)
- \- Should issue debt correctly
- ✓ Should withdraw collateral amount correctly (1345ms)
- ✓ Should burn debt correctly (626ms)

SyntheX

Initialization
- ✓ Should initialize correctly

Setters
- ✓ Should set address (44ms)

Pause/unpause
- ✓ Should pause and unpause contract (70ms)
- ✓ Shouldn't pause and unpause contract by everyone but L2Admin (44ms)
- ✓ Shouldn't call specified any function when paused (63ms)

Rewards operations

- ✓ Should set pool speed with adding reward token to list (75ms)
- ✓ Should set pool speed without adding reward token to list (104ms)
- ✓ Shouldn't set pool speed if reward token already added but param addToList is true (109ms)
- ✓ Should remove reward token from list (100ms)
- ✓ Should update pool reward index (135ms)
- ✓ Should claim rewards (1210ms)
- ✓ Should claim rewards with 2 pools (collecting rewards from a pool they don't belong to) (4243ms)
- ✓ Should claim rewards with 2 pools (check rewards calculation) (3437ms)

SyntheXToken

Initialization

- ✓ Should correctly initialize SyntheXToken contract

Minting

- ✓ Should allow L1Admin to mint tokens
- ✓ Should not allow non-L1Admin to mint tokens

Pause/Unpause

- ✓ Should allow L2Admin to pause and unpause the contract (77ms)
- ✓ Should not allow non-L2Admin to pause and unpause the contract (40ms)
- ✓ Should not allow transfers when paused (103ms)

119 passing (29s)

TEST COVERAGE RESULTS

FILE	% STMTS	% BRANCH	% FUNCS
SyntheXToken.sol	90	62.5	83.33
EscrowedSYX.sol	100	81.03	95.83
Crowdsale.sol	96.97	83.87	93.75
BaseTokenRedeemer.sol	96	54.55	83.33
SyntheXStorage.sol	100	100	100
SyntheX.sol	97.3	70.45	95.24
AddressStorage.sol	100	100	100
PoolStorage.sol	100	100	100
Pool.sol	93.84	59.86	87.5
ERC20X.sol	89.47	64.58	84.62
All files	**96.36**	**77.68**	**92.36**

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.