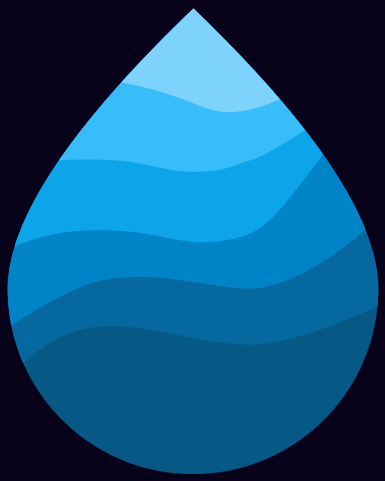


LIQUISTAKE
SMART CONTRACTS AUDIT

zokyo

February 8th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING LIQUISTAKE

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- Critical issues: 0 issues found.
- High issues: 3 issues (resolved): 0 points deducted.
- Medium issues: 4 issues (3 issues resolved): -5 points deducted. While the unresolved issue has no significant impact, it may grow as the total supply of pool is increased. The LiquiStake team has acknowledged the issue and verified that it will be resolved in the future.
- Low issues: 5 issues (4 resolved, 1 acknowledged): -1 points deducted per the concern regarding the infinite approval, which is considered as violation of secure token operations.
- Informational issues: 14 issues (resolved or verified): -4 point deducted based on the present race condition risk, heavy dependence on 3rd party protocol, its settings (such as withdraw delay) and performance of a sole validator.

Thus, 100 - 5 - 1 - 4 = 90.

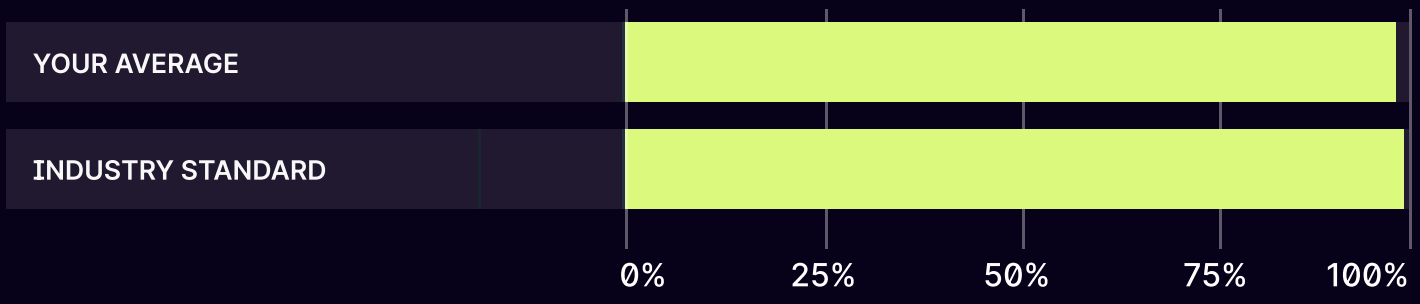
TECHNICAL SUMMARY

This document outlines the overall security of the LiquStake smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the LiquiStake smart contracts codebase for quality, security, and correctness.

Contract Status

Testable Code



99% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the EVM network's fast-paced and rapidly changing environment, we recommend that the LiquiStake team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the repository:
https://github.com/liquistake/smart-contracts
Initial commit. main branch: 014bfa8d0071f566fb3a87cea95159ab01a9816b
Final commit, main branch: 4a7179336cb1a2c9dd3c876010ce0dcece64cc85

After the audit, the team migrated the code base into the public repository:
https://github.com/liquistake/liquistake-v1
master branch: 0fac976791ea4fe2385f765a10897dbb13cba2d9

No additional changes were performed.

Within the scope of this audit, the auditors reviewed the following contract(s):

- StWSX
- WstWSX
- IStWSX

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most resent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of LiquiStake smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contract by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

Zokyo Security team has conducted an audit for the LiquiStake protocol. LiquiStake is a liquid staking pool built on SX Network's native delegated Proof-of-Stake consensus. It allows users to participate in a liquid pool by staking SX tokens and receiving stWSX shares in return. The LiquiStake then stakes tokens under a certain immutable validator on behalf of users. The scope of the audit included 3 smart contracts:

- IStWSX - abstract ERC-20 implementation which supports shares mechanism;
- StWSX - core smart contract of the protocol that represents the liquid pool. It handles deposits, withdrawals, and rewards compounding and distribution;
- WstWSX - Wrapped ERC-20 token for stWSX shares.

The audit's goal was to analyze the implementation of smart contracts for common vulnerabilities, especially those connected to liquid staking, check their compatibility with SX Network and its native staking, and validate the security of users' funds.

Auditors have found 3 high-risk, 4 medium-risk, several low-risk, and informational issues. The first high-risk issue was connected to the inaccurate handle of slippage during rewards collection. The issue might lead to a front-run attack due to the nature of the compound mechanism that swaps reward tokens to the SX token via the decentralized exchange. The second issue was connected to the possibility of updating core parameters such as SX token, validator address, and proxy without a proper migration mechanism. The third issue was connected to the inaccurate calculation of fee shares during reward collection, which could lead to the loss of the protocol fees. Medium-risk issues were connected to the absence of adjustment of an initial number of oracles, absence of reward amount validation, possible mismatch of balances and total supply, and incorrect order of operations in the reward collection function. While all the issues were fixed or verified, the Medium-3 issue remained unresolved. It is connected to the problem with dust-level discrepancies after rewards claiming between the total supply and users' balances (based on the shares-to-token conversion). The LiquiStake team has acknowledged the issue and assured auditors that the team will apply the offchain tracking of "dust rebates." Although the discrepancy has dust-level and auditors tested it against the influence on the protocol (which appeared to be negligible), there is a risk of its accumulation after a prolonged period.

It should be noted that the protocol heavily relies on the native staking of SX Network. Any changes in the staking can affect the ability of the LiquiStake protocol to operate. The protocol also relies on the performance of the validator to which SX tokens are delegated. Since the validator is immutable, it is highly recommended that LiquiStake use only trusted or validator nodes.

Zokyo Security team has prepared a suite of unit tests in order to check the correctness of smart contracts' implementations. During testing, auditors verified the basic logic of the protocol, its compatibility with the SX Network staking, shares calculations, and rewards distribution. Additionally, auditors have prepared scenario tests to verify any potential threats. The list of scenarios includes:

- The common flow of the protocol, including testing the full cycle of the protocol, starting with deposits, then testing rewards distribution and finishing with the unstaking;
- Double conversion of WSX shares during unstaking;
- Rounding error for balances and total supply;
- Rounding error during shares transferring;
- Inflation attack;
- Possible loss of funds for late users.

All potential threats were not confirmed except for the rounding error for balances and total supply, which is described in Medium-3. The Zokyo Security team prepared all the tests presented in the report.

The total security level of the protocol is sufficient. Contracts are well-written and have good natSpec documentation. Though the stWSX smart contract relies on 3rd party protocol and the performance of the validator, auditors recommend that the LiquiStake team monitor the performance of SX Network staking and implement a policy for cases when migration of funds is required.

Roles and Responsibilities

stWSX.sol

1. Admin

The admin role is maintained via AccessControl by OpenZeppelin. The role for admin is "ADMIN_ROLE," which is granted to the msg.sender during deployment. Admin is in charge of adding/removing new admins and oracles. Furthermore, the admin can update all the settings of the smart contract, including addresses, fee parameters, and slippage. Additionally, admin can force unstake and withdraw unstaked operations for cases when oracle is experiencing any troubles.

2. Oracle

Oracle is maintained via AccessControl by OpenZeppelin. Oracles are in charge of executing the functions for unstaking WSX from protocol and claiming rewards. Oracle's responsibility is to call functions on time and pass correct amounts. Otherwise, the protocol's storage may be corrupted, or users' funds may be stuck until Oracle executes the necessary functions.

3. Regular user

Users are the main actors of the protocol who perform deposit/unstake/claim of WSX tokens. In order to do so, users have to approve the SX token in the quantity that they want to deposit and initiate an unstake procedure when they wish to claim their initially deposited SX + accrued rewards.

4. DAO

DAO is used as an escrow address, which receives fees during deposits and rewards reporting.

5. Validator

The validator plays a crucial role as users delegate their stakes under the specified validator. The validator's performance then affects the rewards accrual reported by Oracle.

Roles and Responsibilities

WstWSX.sol

1. stWSX

stWSX is an instance of StWSX.sol and is used as an original ERC-20 token, which can be wrapped.

2. Regular user

Users are the main actors who can wrap and unwrap their stWSX tokens.

Note: Wrapping contract will not accept stWSX sent directly to the contract, thus any stWSX sent directly will effectively become stuck.

Settings

stWSX.sol

1. Mint Fee

The Mint Fee is used for calculating fees during deposits. Fees are sent to DAO. Set to 1% during deployment. It can be set to any value up to 35%. It can be set by admin.

2. Reward Fee

The Reward Fee is used to calculate fees during the collection of staking rewards. Fees are sent to DAO. The initial value for the deployment is 10% but can be changed up to 35%. It can be set by admin.

4. Staking Proxy

Staking Proxy is the address of SX Staking. It is set during deployment and remains immutable.

5. Compound Stake Proxy

Compound Stake PRoxy is the address of the Compound Stake in the SX Network. It is set during deployment, but the admin can change it if the SX Network changes.

Settings

stWSX.sol

6. WSX Token

WSX Token is the address of the ERC-20 token used by SX Token for staking. It is set during deployment and remains immutable.

7. DAO Address

The DAO Address is a treasury that receives protocol fees. It is set during deployment and can be set by the admin.

8. Validator address

The Validator address is used for delegating all the stakes in SX Staking. It is set during deployment and remains immutable.

List of valuable assets

stWSX.sol

1. SX tokens

Though SX tokens are not stored on the contract's balance after deposit, they are staked from the stWSX, making it the owner of staked SX. Moreover, unstaked SX is indeed stored on the contract's balance until claimed by users.

2. st WSX shares

stWSX shares are minted to users during staking of SX tokens. They represent users' position and allow them to withdraw deposited SX and accrued rewards. Shares act like standard ERC-20 tokens, meaning they can be transferred, approved, traded, etc.

3. Reward tokens

Since SX Staking provides additional rewards, they are also earned by stWSX. Rewards are not stored on the stWSX balance but are compounded during rewards collection.

Settings

WstWSX.sol

1. st WSX

stWSX is an original ERC-20, which can be wrapped or unwrapped.

7. Wst WSX

Wst WSX is a wrapped representation of ERC-20 stWSX tokens.

Note: No contract has rescue functions, thus any asset sent directly to the contract will effectively become stuck.

Deployment script

stWSX.sol

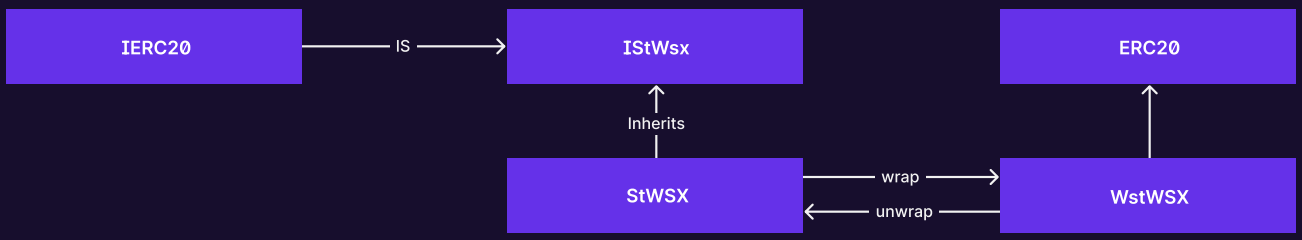
Deployment script is located at ./scripts/deploy.ts. The file contains the deployment instructions for stWSX.sol. The script is currently targeted for SX Testnet only, as it uses the addresses of contracts in the SX Testnet.

Potential threats

1. Contracts are heavily dependent on 3rd parties which are currently out of the scope, thus their behavior cannot be validated.
   - Oracles (making the role of automated workers) cannot be validated for their operating schedule, the correctness of the order of operations, or honesty regarding the reporting of rewards and claimed amounts. And the fact of actual WSX transferring to the contract.
   - StakingProxy cannot be validated for correct reward distribution from the validator. This is especially crucial as the current contract contains managing functions that wrap calls to stakingProxy for admins. Therefore, the flow, where the user can directly access stakingProxy, cannot be validated - e.g., for force unstake/withdraw scenarios.
2. Autocompounding cannot be properly verified as it is fully performed on 3rd party contracts.
3. Protocol depends on the performance of the validator. In case on misbehaving of the validator, the users of the protocol may end up by losing their funds. Since validator is immutable it is recommended that the LiquiStake delegates funds only to the trusted validator. It is also recommended to prepare a policy for a quick replacement of validator or distribution of funds across several validators and implement a migration mechanism based on this policy.

LiquiStake scheme

StWSX Overview



StWSX is **custom** ERC20 implementation

name: Liquid Staked WSX
symbol: stWSX
decimals: 18

totalSupply: equals totalPooledWSX, all WSX amount which:
- is deposited by user via deposit() function' and is NOT currently requested for unstake
- OR is staking reward already reported by Oracle after claim from staking
- Also, does not include fee deducted and sent to DAO

balanceOf: StWSX token by nature is based on shares the user has in the liquid staking pool. Though, by design, shares should represent the part of the pool, and balance should reflect the amount of WSX the user owns in the pool. So, balance is calculated as:
shares[user] * totalPooledWSX / totalShares

Shares info:

getTotalShares() - sum of shares of all users
sharesOf(account) - shares for a certain user

Convertors
getSharesByPooledWSX(wsxAmount)
shares = amount * totalShares / totalPooledWSX

getPooledWSXByShares(sharesAmount)
amount = shares * totalPooledWSX / totalShares

ERC20 interface
approve() / allowance / increaseAllowance / decreaseAllowance() - set of allowance functions work with **token** reflection (pre-calculated from shares virtual **WSX amount**)

transfer()
trasnferFrom()
- take **amount in WSX**
- **convert it to shares**
- change balance of **shares** between the sender and recipient
- transferFrom() checks and spends allowance in tokens

transferShares(): strictly changes balance of shares between sender and recipient

transferSharesFrom():
- **converts shares to tokens**
- checks allowance in tokens
- spends allowance in tokens
- changes balance of **shares** between the sender and recipient

_mintShares()
_burnShares()

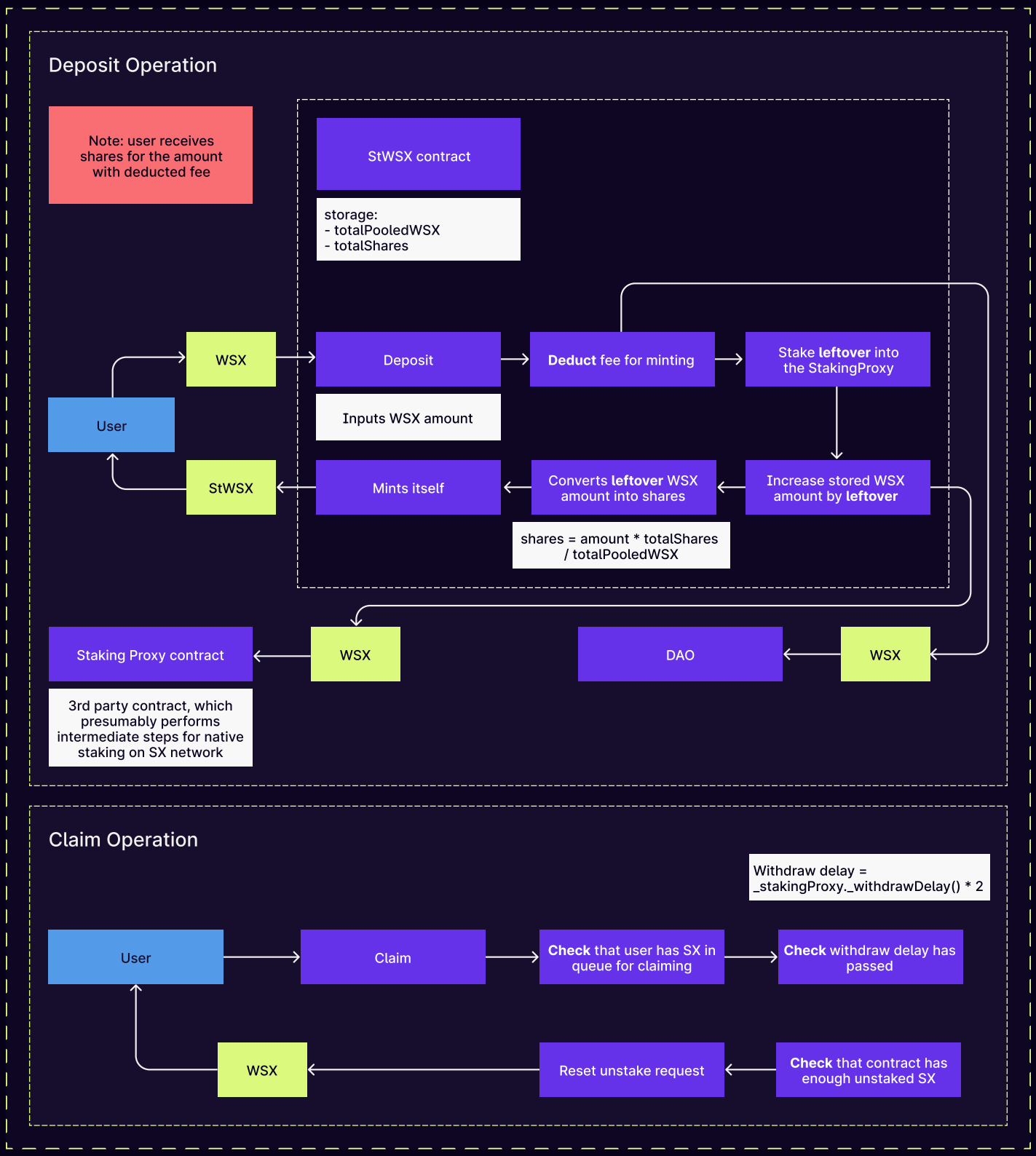
Internal functions to change the balance of shares without changing stored WSX amounts. Work as mechanisms for inflation (mintShares) and deflation (burnShares) of the shares price in WSX → thus making shares reflect less or more WSX from the pool

deposit()
unstake()
claim()
User's interface for interaction with liquid staking (staking / withdrawing WSX)

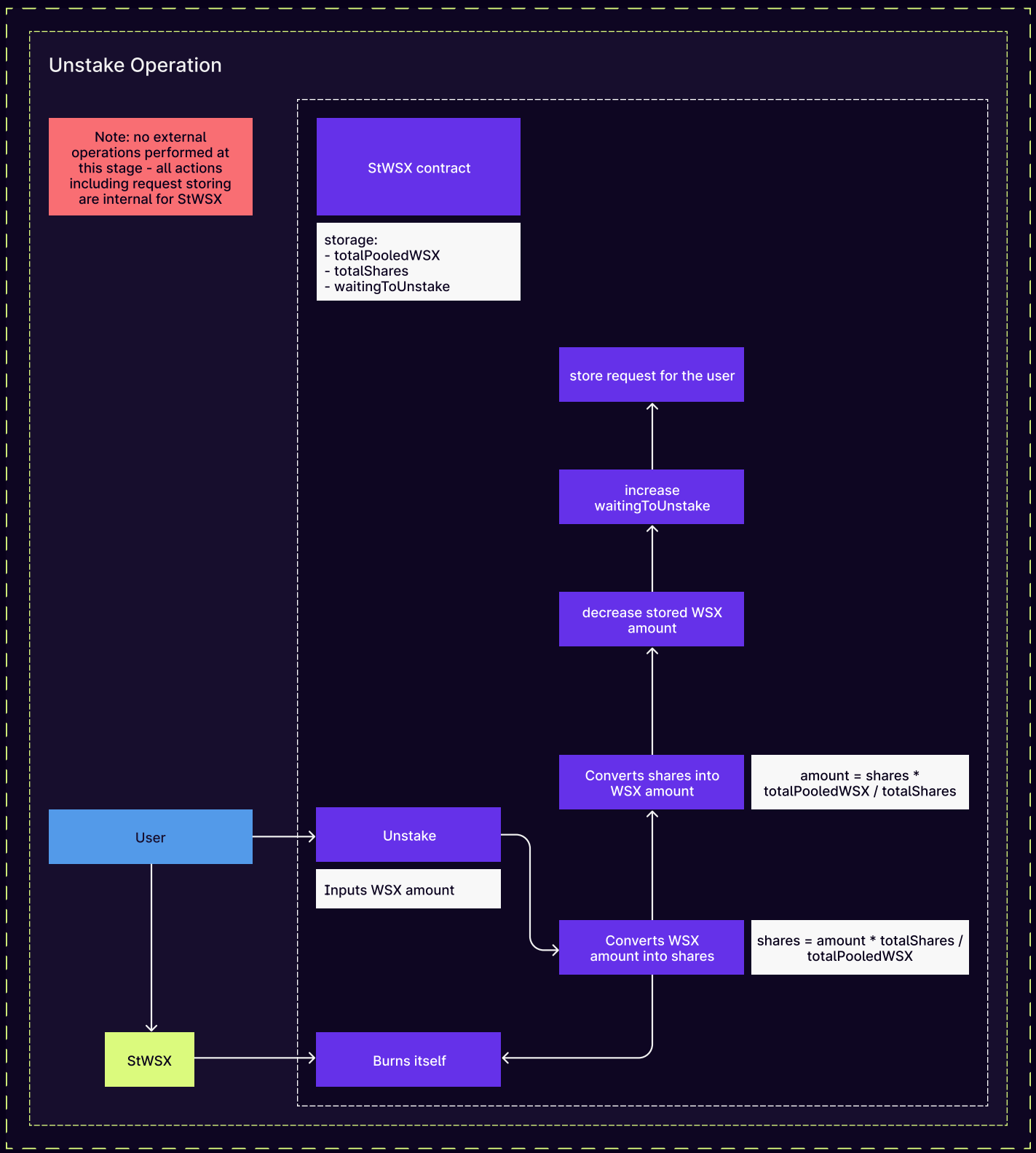
Oracle functionality
- responsible for requesting unstake from StakingProxy
- responsible for notifying StWSX about new rewards
- responsible for compounding rewards into WSX
- responsible for claiming unstaked WSX from StakingProxy to StWSX

Admin functions
- can change any parameter of the pool

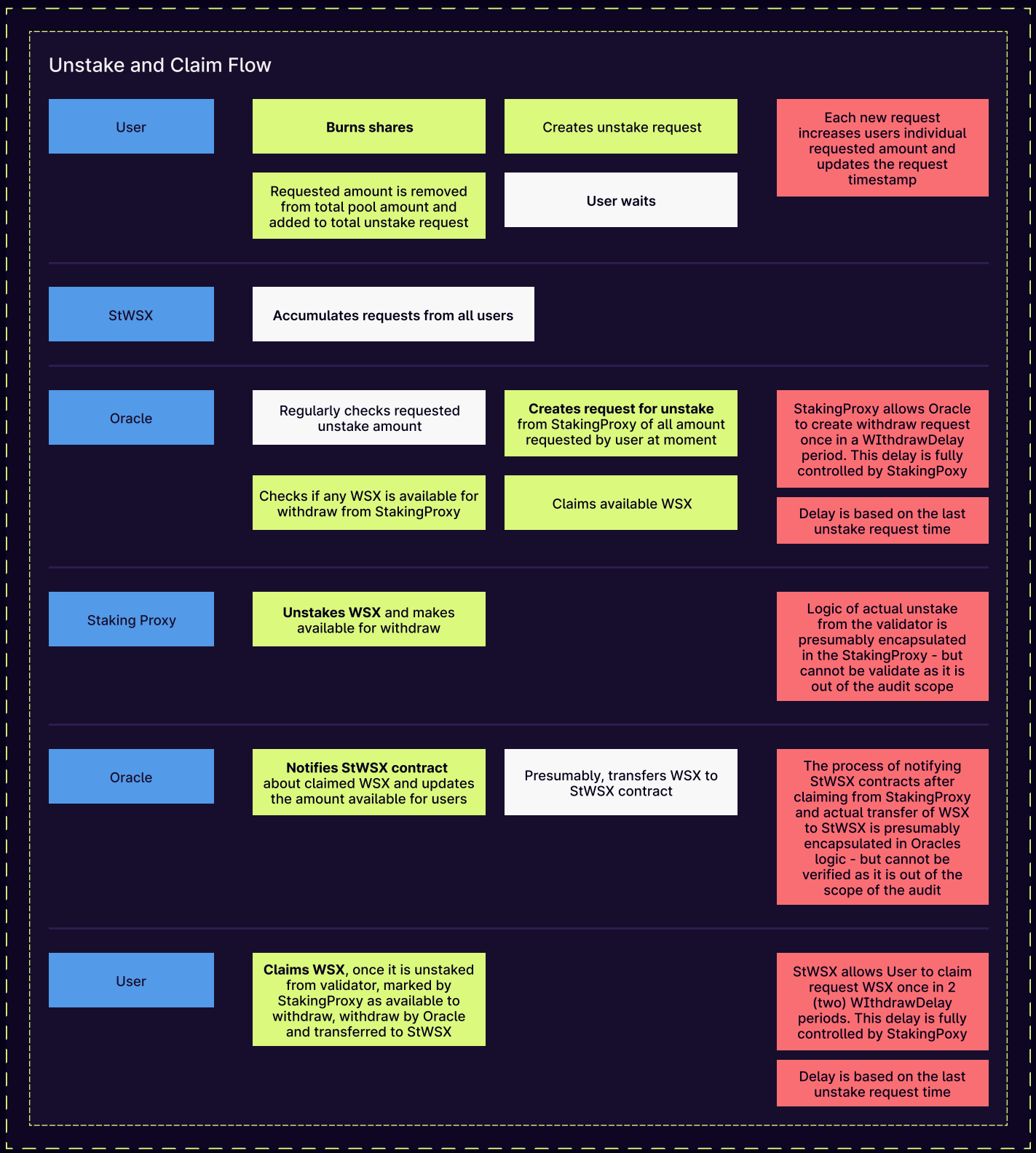
LiquiStake scheme



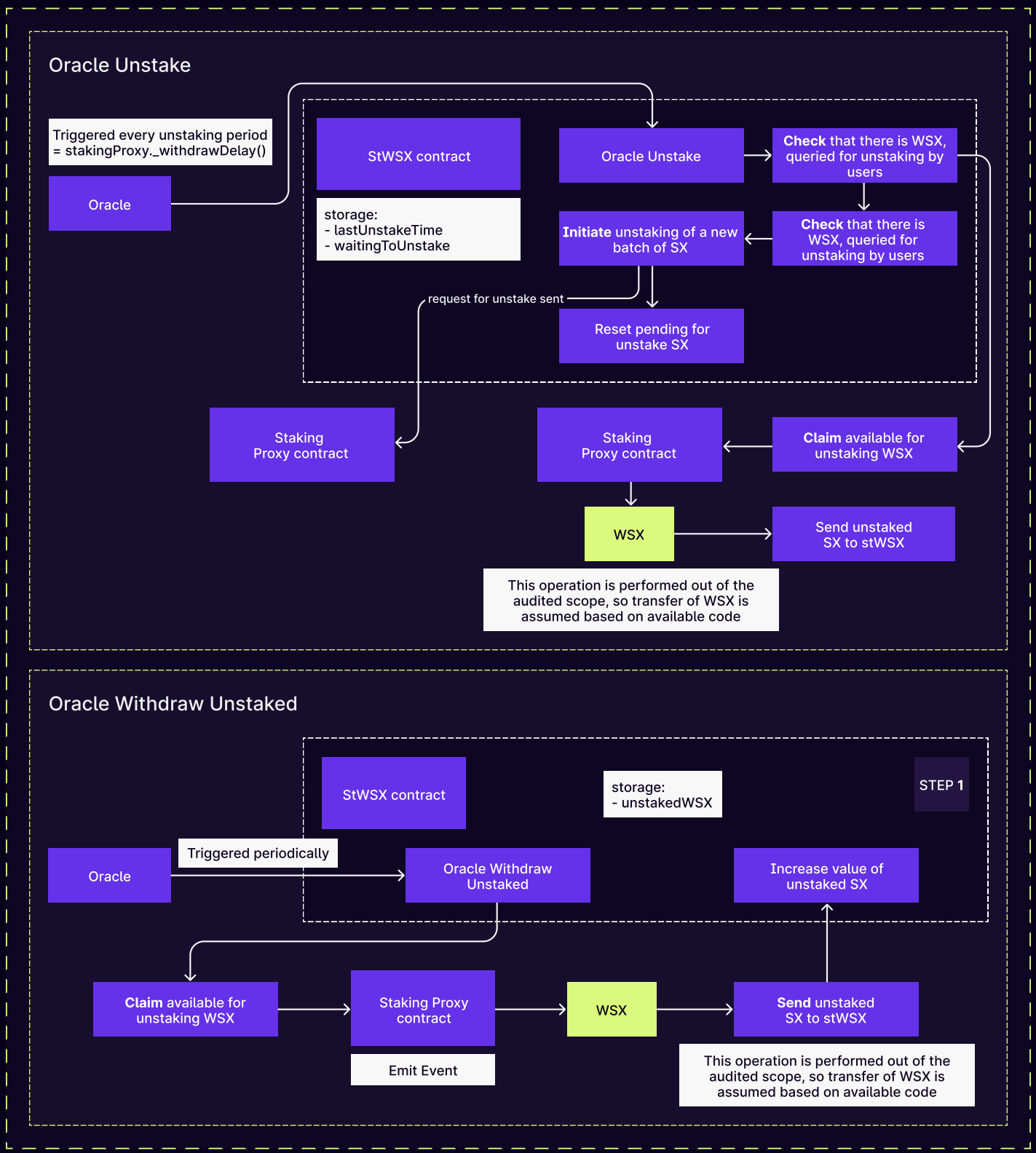
LiquiStake scheme



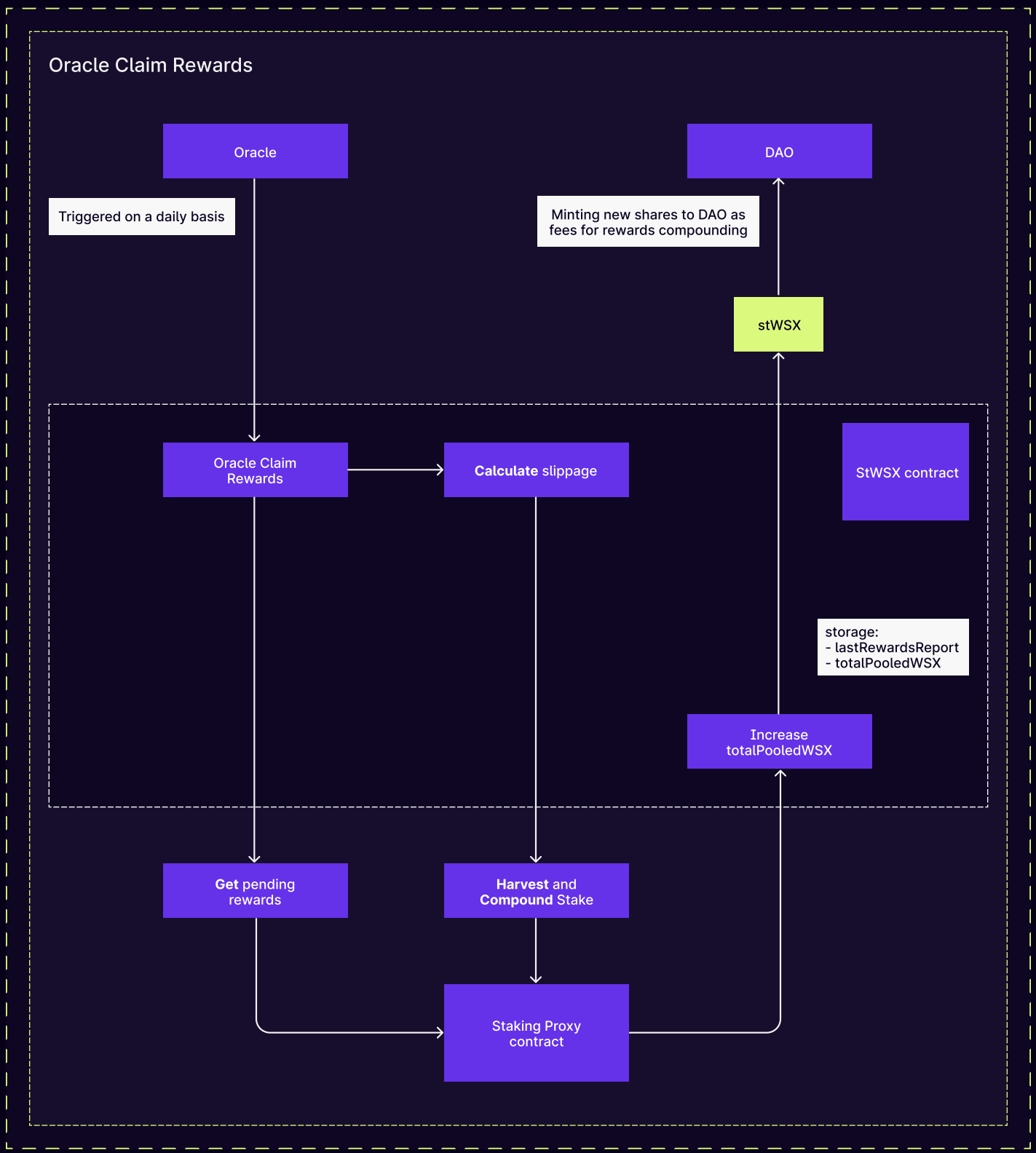
LiquiStake scheme



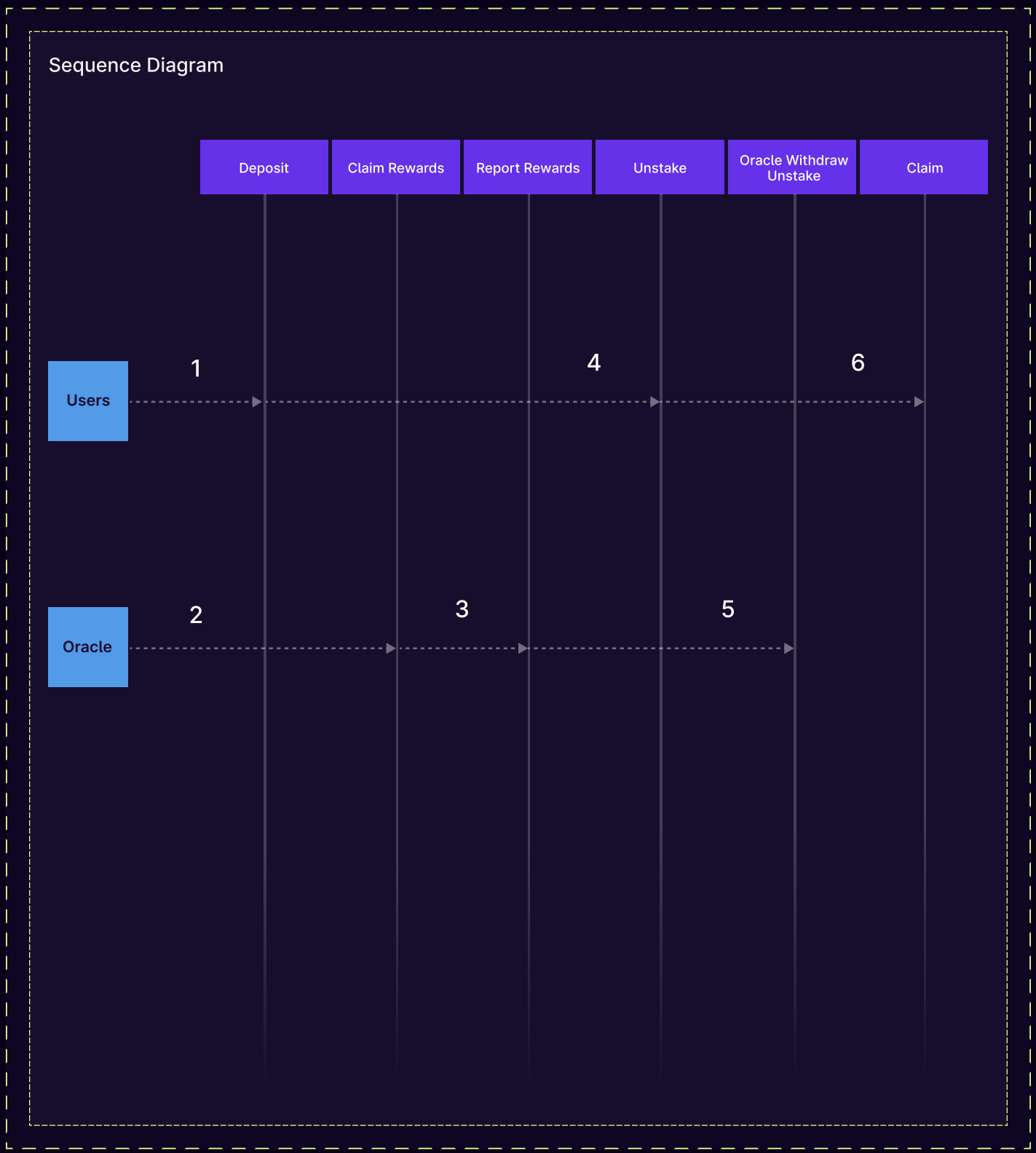
LiquiStake scheme



LiquiStake scheme



LiquiStake scheme



STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

HIGH-1 | RESOLVED

Incorrect handling of slippage for rewards collection.

StWSX.sol: oracleClaimRewards(), line 332.
Currently, slippage is calculated as a certain percent from accrued reward and is stored in the array "amountOutMins". However, based on the logic of CompoundStake.sol, values "amountOutMins" are used as minimal amounts of WSX, which will be received through exchanging accrued rewards into WSX via the SharkSwap exchange. Thus, storing values representing the reward token's percentage may lead to reverting the swap since SharkSwap will compare the amountOutMin with WSX, not the reward token.
Additionally, calculating slippage inside the smart contract before the swap will not protect against the frontrun attack and may lead to the loss of rewards. That is why the most optimal way of handling slippage is passing it as a function parameter and calculating it on the backend part in advance.
Recommendation: Calculate slippage as an amount of WSX expected to be received for the accrued reward amount. Pass an array with slippage values as a function parameter, calculating it on the backend in advance to protect against possible frontrun attacks.

Post-audit. The reward token list and exchange slippage are now passed as function parameters. According to the LiquiStake team, slippage will be calculated on the backend. Functionality connected to the slippage was removed from the contract.

HIGH-2 | RESOLVED

Updating of WSX, Staking Proxy or Validator may lead to the loss of funds.

StWSX.sol: oracleClaimRewards(), line 332.
StWSX.sol: setWSXToken(), setStakingProxy(), setValidatorAddress().
Updating WSX, Staking Proxy, or Validator may cause users whose funds were not unstaked before the update to lose them. Thus, when updating these parameters, users' staked funds may get stuck.
Recommendation: Implement a migration mechanism for staked WSX when updating the core parameters.

Post-audit. Setters were removed so that WSX, Staking Proxy, and Validator parameters can only be set during deployment. The migration will be performed via redemption and deposit to a new contract.

HIGH-3 | RESOLVED

Inaccurate calculations of fee shares for DAO.

StWSX.sol: oracleClaimRewards().
When rewards are claimed, part of them are allocated to the DAO in the form of shares as protocol fees. However, the fees are not calculated accurately since the whole amount of the `reward amount` is added to the `totalPooledWSX` before the calculating and minting of fees. As a result, the DAO will actually receive less when shares are redeemed. And in case `rewardAmount` is added to the `totalPooledWSX` after minting, DAO will receive more, cutting user rewards.
Recommendation: Before minting shares, add only the reward amount that should be distributed to users (rewardAmount - rewardFeeAmount) to the `totalPooledWSX.` The rest (rewardFeeAmount) should be added after minting.
Example of the code:

```
totalPooledWSX += (rewardAmount - rewardFeeAmount);

uint rewardFeesShares = getSharesByPooledWSX(rewardFeeAmount);
_mintShares(DAO_ADDRESS, rewardFeesShares);

totalPooledWSX += rewardFeeAmount;
```

Post-audit. Changes were applied.

MEDIUM-1 | RESOLVED

Initial number of Oracles is not adjusted.

StWSX.sol, constructor
The initial Oracle is assigned the Oracle role, but the number of Oracles (numOracles) is not adjusted.
Recommendation: Add increment of numOracles to the constructor.

MEDIUM-2 | RESOLVED

No validation for WSX rewards / unstaked amount.

StWSX.sol, oracleReportUnstakedWithdraw(), oracleReportRewards()
It is assumed that the necessary amount of WSX will already be present on the contract at the moment of reporting. However, there are no checks to show that the rewards / unstaked amount was actually transferred to the contract - either by Oracle or by another entity.
Recommendation: Add validation for the balance before and after reporting and/or add transferFrom() (or another hook) for WSX into the reporting functions. Auditors assume that such a check may also be added to the withdraw/claim functions, depending on the process of forwarding tokens from the StakingProxy.

Post-audit. The process of processing the rewards collection and unstaking was updated as the auditors have suggested (See Info-12).

MEDIUM-3 | UNRESOLVED

Sum of balances doesn't match up with total supply after rewards collection.

StWSX.sol
After the claiming of rewards with the function oracleClaimRewards(), the total supply and sum of users' balances don't match up. The difference is estimated in several wei only. However, it has the potential to increase as deposit balances and rewards increase. Moreover, users are able to specify more tokens than their balance when unstaking. However, the amount by which they could specify more doesn't exceed the difference between the sum of balances and total supply. If all the users and DAO unstake their shares, the total supply will remain non-zero.
The issue is marked as Medium since the round of testing already showed that balances and total supply may differ, and this difference will increase as deposits and rewards increase.
Recommendation: Increase the accuracy of calculations when calculating shares.

Post-audit. The LiquiStake team has acknowledged the issue and decided to implement a back-end service that will track the difference and perform a "dust rebase" on smart contracts. However, the fix wasn't implemented during the current audit iteration.

MEDIUM-4 | RESOLVED

Updating the total supply before minting of new shares.

StWSX.sol: oracleClaimRewards().
When collecting rewards, part of them is transferred to the DAO in the form of new shares. However, the shares are calculated and minted after the variable `totalPooledWSX` is increased (Contrary to deposit flow, where shares are calculated and minted first and the `totalPooledWSX` is increased after). Though the current tests showed that the change of order may increase accuracy up to several weis only, this can be increased as the number of rewards increases.
Recommendation: Change the order of minting shares and increasing `totalPooledWSX`, so that it corresponds to the deposit flow.

Post-audit. The order of operations correspond to the deposit flow now.

LOW-1 | RESOLVED

No events for critical information.

StWSX.sol: oracleUnstake(), forceUnstake(), forceWithdrawUnstaked(), oracleWithdrawUnstaked(), oracleClaimRewards(), oracleReportUnstakedWithdraw(),

Mentioned functions require events to show the change of the requested amount. For now, the contract does not log either the receiving of WSX from StakingProxy or the moment of fulfilling requests and re-setting of the waitingToUnstake variable and amount of the request. And so, the critical information cannot be retrieved in any other way than by reading the rough storage changes.
Recommendation: Add event tracking the moment of re-setting of the accumulated unstake request, amount, and responsible Oracles.

LOW-2 | VERIFIED

Infinite allowance.

IStWSX.sol, INFINITE_ALLOWANCE, transferFrom()
By default, the contract allows user flows with an infinite allowance, which is, in general, bad practice. It creates a greater risk for users in case of protocol compromise or user misbehavior during interactions with the dApp.
Recommendation: Remove the default flow with infinite allowance and ensure that the UI has appropriate notification regarding the allowance policy and requests only strictly necessary amounts from users for approval.

Post-audit. According to the Liqui Stake team, the functionality will remain for ease of use. Also, according to the dev team, infinite allowance is not used on the frontend part. Nevertheless, the security team leaves the concern regarding the existence of the infinite approve which increases the risk for users.

LOW-3 | RESOLVED

Magic number.

StWSX.sol, lines 418, 428, 438
The contract utilizes a magic number (`10000`) representing the precision of the percent calculation. In general, it is a bad practice, which decreases the readability of the code and may affect further development.
Recommendation: Consider usage of the constant for precision.

LOW-4 | RESOLVED

Missing value checks

1. StWSX.sol: setMaxSlippage().

The parameter is not checked for a minimum value. This allows slippage to be set to `0` or a very low value, allowing it during reward harvesting.

2. StWSX.sol: setMintFee(), setRewardFee().

Fee parameters are not checked for a maximum value. This makes it possible to set fees to a large percentage (up to `100%`), taking all the deposits from users or rewards.

Recommendation: Add validations for slippage and fees.

Post-audit. After a series of fixes, the validations were properly implemented. The LiquiStake team has removed setMaxSlippage() function due to its uselessness. As for the functions setMintFee() and setRewardFee(), a 35% limitation was introduced, allowing to set zero fees as well.

LOW-5 | RESOLVED

Missing default visibility.

StWSX.sol
DAO_ADDRESS
totalPooledWSX
Visibility of variables should be explicitly marked, in order to increase the readability of the code.

Issue is marked as Low-risk, as it is classified as standard Solidity vulnerability
Recommendation: Explicitly mark visibility of variables.

Post-audit. Visibility is now explicitly marked for all the variables.

INFORMATIONAL-1 | RESOLVED

Increase/Decrease allowance functionality.

IStWSX.sol
Functions for increasing/decreasing allowance are substandard and usually not recommended for usage, as they amplify the risks connected to incorrect and dangling allowances. For example, OpenZeppelin removed them from their ERC20 implementation starting from OZ 0.5.x.
Recommendation: Consider removing of mentioned functions and rely on best practices of approve usage in case of required allowance change: approve(N) → approve(0) → approve(M).

Post-audit. The contract's interface corresponds to the interface of IERC20 of OpenZeppelin 0.5.x version. Increase/Decrease allowance functionality was removed.

INFORMATIONAL-2 | RESOLVED

Unconventional usage of AccessControl.

The current implementation of StWSX.sol bypasses the DEFAULT_ADMIN_ROLE and grantRole() mechanics in favor of a custom role distribution interface.
Recommendation: Verify the current access control implementation usage or consider the conventional flow usage.

Post-audit. After a series of fixes, the roles management functionality was properly implemented. DEFAULT_ADMIN_ROLE is used now and is granted in constructor. Functions grantRole(), revokeRole() and renounceRole() were overridden to contain necessary validations.

INFORMATIONAL-3 | RESOLVED

Testnet addresses are hardcoded in the constructor.

StWSX.sol, storage and constructor()
Currently, the contract utilizes testnet addresses for 3rd party contracts, dao and Oracle, which are hardcoded directly in the constructor. This approach may not be flexible and convenient when migrating to the mainnet and may lead to errors. Also, the correctness of current addresses should be verified.

Though the LiquiStake team left the acknowledging comment in the code, it may still affect the mainnet deployment and should be noted in the report - especially since the repo does not contain deployment scripts.

Recommendation: Pass addresses as constructor parameters. Prepare deployment scripts for testnet and mainnet in advance. Verify the correctness of current addresses in the testnet.

Post-audit. 3rd party addresses were moved to the constructor parameter. Deployment script was added for the testnet deployment.

INFORMATIONAL-4 | RESOLVED

No events during changes of the critical information.

StWSX.sol: setMintFee(), setRewardFee(), setMaxSlippage(), setStakingProxy(), setCompoundStakeProxy(), setWSXToken(), setDAOAddress(), setValidatorAddress()

StWSX: claim(), unstake()
WstWSX.sol: wrap(), unwrap().

Setters should emit events in order to keep track of historical changes of the variables and user's actions.
Recommendation: Consider adding events, so the historical info will be available for the dApp users.

INFORMATIONAL-5 | RESOLVED

Gas optimization suggestions.

1. StWSX.sol: deposit, line 129.

Function deposit() contains a redundant check for an amount greater than `0`. The validation is redundant since the next validation checks that the amount is greater than the minimum deposit amount, automatically excluding the possibility of the amount being equal to `0`.
Recommendation: Remove redundant validation.

1. StWSX.sol: event FeesCollected, parameter timestamp.

The parameter `timestamp` is excessive and increases the gas cost for event emitting since the timestamp can be obtained from the block in which the event was emitted.
Recommendation: Remove excessive parameter.

1. StWSX.sol: unstake(), line 240.

The function uses operator += to add requested amount to `_unstakeRequests`. Instead of +=, operator = can be used, since _unstakeRequests is checked to be `0` in line 222.
Recommendation: Change += to =.

1. Redundant initialization.

StWSX.sol: line 46, 65, 68, 71.
Variables are explicitly initialized with `0`, though, all uint variables are initialized with `0` by default.
Recommendation: Remove explicit assigning to `0`.

INFORMATIONAL-6 | VERIFIED

Potential optimization for IStWSX.

IStWSX.sol, _emitTransferAfterMintingShares()
Function _emitTransferAfterMintingShares() seems over-engineered, as it is a particular case of the _emitTransferEvents() function, and can be eliminated, thus simplifying the code.
Recommendation: Consider optimization of _emitTransferAfterMintingShares() function.

Post-audit. The LiquiStake team has decided to leave the implementation as is for simplification of logic understanding.

INFORMATIONAL-7 | VERIFIED

Possible manipulation with rewards period length.

StWSX.sol, oracleReportRewards()
Rewards reporting logic emits events with reward period length. However, there are no control methods to validate if oracles behave honestly - as for now periodLength can be arbitrary. For example, oracle may withhold the transaction that should be submitted for accounting to prolong the periodLength or submit the next one within a shorter period. Thus makes possible manipulation with the periodLength, which may be crucial for the accounting further in the dApp internal flow.

The issue is marked as Info, as it is connected to the business logic and operations outside of the contract logic. Thus, it cannot be classified without comments from the LiquiStake team.
Recommendation: Verify the reward reporting flow and significance of the reward period length information for the dApp.

From client: According to the Liqui Stake team, the period is used for APR calculations only. The calculation does not impact the functionality or actual APR provided by the contract and only serves as historical data of provided rewards.

INFORMATIONAL-8 | RESOLVED

Potential loss of accuracy

StWXS.sol, unstake()
The interface suggests that the user enter the amount he wants to withdraw. However, since the accounting is performed in shares, the unstake function performs a double conversion, leading to accuracy losses.
The issue is marked as Info, as it is on the list for confirmation during the testing stage.
Recommendation: Consider adding interface for operation in shares inputs.

Post-audit. The testing phase didn't show a loss of accuracy during the withdrawal operation. However, the security team decided to leave it as an informational note, as double conversion creates a slight risk of dust problems during further development.

User's funds may be stuck on the contract because of race conditions.

StWSX.sol, claim() (Verified)
Users can wait a long time until they have the chance to withdraw the whole requested unstake amount, as by the contract's design, the possibility of the unstake is bound to the availability of WSX on the contract. Also, by design, the contract provides a pooled approach without diversification between users. Also, the contract does not have a WSX tracking mechanism (issue Medium-2). So, it is possible a situation where a user requested an unstake, then several more users requested an unstake and claimed rewards (after appearing on the contract) before the first user could, thus leaving him waiting for the next rewards round. All this time, user tokens are locked on the contract, waiting for the appropriate timeslot with enough WSX available on the contract.

Thus, users become vulnerable to race conditions, and some may have funds stuck for a long time (until enough WSX is available on the contract)

The issue is marked as Info, as it refers to business logic decisions and cannot be classified without comments from the LiquiStake team.

Recommendation:

- At this point, auditors understand that by design, the contract supports a pooled approach and cannot support user diversification. Still, the first recommendation is to provide such - e.g. via additional storage structures tracking the time of unstakes, and not allowing "next wave" users to claim before previous ones (with some appropriate waiting time limit, of course, for the first users)
- Since the contract already provides a pooled requests model, the same "pooled claim" model may be applied for claims - separating users into "claiming waves", ordering claims, and eliminating race conditions (again, with an appropriate waiting time limit to prevent the previous wave from blocking out the next one)
- The dApp UI interface must clearly reflect the possibility of race conditions and the absence of an exact waiting time for claiming availability.

Auditors have prepared a scenario for the race conditions:

1. The first user requests unstake.
2. Oracle finalizes unstake request, so the first user is able to call the claim.
3. The second user requests unstake.
4. The second user claims.
5. The first user is not able to claim and should wait for another Oracle finalization since another user has used the WSX tokens requested to satisfy the claim. In this case, the StSFX contract throws an error message stating `Not enough WSX to satisfy claim.`

From client. The Liqui Stake team has verified that this won't be an issue in the mainnet where the withdrawal period will be sufficiently long. Still, auditors leave a concern regarding the race condition available in the smart contract.

INFORMATIONAL-10 | VERIFIED

Autocompounding is completely independent from the StWSX functionality.

StWSX.sol, oracleClaimRewards()
Autocompounding flow is fully implemented on 3rd party contracts and components, has no effect on contract storage, and does not use any contract data. It is also crucial, as the autocompounding flow may leave dust on the StWSX contract with no ability to rescue it. Not saying about its own independent issues (see High-1).

Issue is marked as Info, as it refers to the business logic decisions.
Recommendation: Consider removing the autocompouding feature out of the contract into a separate independent one.

From client: The auto compounding functionality will stay due to its significant utility.

INFORMATIONAL-11 | RESOLVED

Typos in NatSpec.

1. WstWSX.tokensPerStWSX()
   The `return` description indicates that it returns the amount of wstWSX for a 1 amount of wstWSX for 1 amount of wstWSX which is a mistake.
2. WstWSX.unwrap()

The `_wstWSXAmount` description has a mistake in the word `uwrap`.
Recommendation: Change the description of the `return` param in function tokensPerStWSX() to: "Amount of wstWSX for a 1 stWSX" and correct the spelling of the word `unwrap` in the unwrap() function.

INFORMATIONAL-12 | RESOLVED

Rewards and unstake flow can be simplified.

The following functions can be merged:

- oracleClaimRewards() & oracleReportRewards()
- forceWithdrawUnstaked() & oracleReportUnstakedWithdraw() + oracleWithdrawUnstaked() & oracleReportUnstakedWithdraw().

The contract utilizes two separate functions to claim rewards and report the increased amount in the Staking Proxy. However, the functionality of functions can be merged since the increase in the staked amount happens during the execution of oracleClaimRewards().
For example, the following code snippet might be used to claim and report rewards:

```
...
uint256 stakedAmountBefore = _stakingProxy.accountStake(address(this));
_compoundStakeProxy.harvestAndCompoundStake(
rewardTokenList,
amountOutMins);
uint256 rewardAmount = _stakingProxy.accountStake(address(this)) - stakedAmountBefore;
oracleReportRewards(rewardAmount);

// continue the logic of Oracle Report Rewards in the same function call.
...
```

As a result, the protocol may exclude the excessive step of listening to StakeCompounded events, maintaining transaction hashes, and calling additional functions. Additionally, this may decrease chances of possible errors in case of misbehaving oracle and writing the wrong amount of reward.

The same happens with the pairs of functions forceWithdrawUnstaked() & oracleReportUnstakedWithdraw() + oracleWithdrawUnstaked() & oracleReportUnstakedWithdraw().

Increasing `unstakedWSX` in a separate function is excessive, since the amount by which the unstakedWSX should be increased can be obtained during the calls forceWithdrawUnstaked() and oracleWithdrawUnstaked() and forceUnstake().

The value can be retrieved in the following way after every call `stakingProxy.withdrawUnstaked()`.
For example:
uint256 pendingUnstakeAmount = _stakingProxy._pendingWithdrawAmounts();
_stakingProxy.withdrawUnstaked();
unstakedWSX += (pendingUnstakeAmount - _stakingProxy._pendingWithdrawAmounts())

The issue is marked as Info since it relies on the out-of-scope implementation of backend Oracle. However, a proposed implementation allows a more secure way of updating the values than relying on an oracle whose actions are not validated on-chain in the smart contract's code.
Recommendation: Merge the functions to eliminate excessive calls for oracles.

Post-audit. Updates were implemented, eliminating excessive calls for oracles.

INFORMATIONAL-13 | RESOLVED

Granting role in revoking role functions.

1. StWSX.revokeRole() In case the role passed as an argument is not the same as the oracle and admin role, the role will be used as a role grant for the user and not the other way around.
2. StWSX.renounceRole() In case the role passed as an argument is not the same as the oracle and admin role, the role will be used as a role grant for the user and not the other way around.

Recommendation: Change grantRole to revokeRole to match the logic of the functions.

INFORMATIONAL-14 | RESOLVED

Unreachable scenario.

StWSX.renounceRole has the check for confirmation that the caller's address and the address that would revoke the role are the same. But in case the role is an admin role, there is a check that the caller should not be the address from which the role will be revoked. In this case, this part of the code is illogical and unreachable.

Recommendation: Remove the block in case a user wants to remove administrator rights from himself and do a revert.

	stWSX WstWSX IStWSX
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	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

Tests written by Zokyo Security

As a part of our work assisting LiquiStake in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the LiquiStake contract requirements for details about issuance amounts and how the system handles these.

IStWSX

✓ After deployment
 # Transfer
✓ Should allow to transfer
✓ Should revert if amount is exceeding balance
✓ Should revert if receiver is zero address
✓ Should revert if receiver is `stWSX` address
 # TransferFrom
✓ Should allow to transferFrom (39ms)
✓ Should revert if amount is approve is exceeding allowance
 # Transfer shares
✓ Should allow to transfer shares
✓ Should revert if amount is exceeding balance
✓ Should revert if sender is zero address
 # Transfer shares from
✓ Should allow to transfer shares from
 # Approve & Allowance
✓ Should allow to approve and get allowance
✓ Should allow to set INFINITY allowance
✓ Should revert if spender is zero address
✓ Should revert if owner is zero address
 # Total supply
✓ Should allow to get amount of tokens in existence
✓ Should allow to get total amount of shares
 # Mint Shares
✓ Should allow to mint shares
✓ Should revert if receiver is zero address
 # Burn Shares
✓ Should allow to burn shares
✓ Should revert if burn account is zero address
✓ Should revert if amount is exceeding shares

StWSX

 # Unstake available time of
✓ Should return correct time (557ms)
 # Unstake request time of
✓ Should return correct time (571ms)
 # Unstake request amount of
✓ Should return correct amount (540ms)
 # Deposit
✓ Should allow to make deposit (552ms)
✓ Should revert when amount to deposit is less than minimum
 # Claim
✓ Should allow to claim (114ms)
✓ Should revert if there is nothing to claim
✓ Should revert if there is not enough WSX to satisfy claim.
✓ Should revert if Insufficient time passed since unstake. (81ms)
 # Unstake
✓ Should allow to unstake (43ms)
✓ Should revert if amount to unstake is zero
✓ Should revert if user has 1 unstake in progress
✓ Should revert when amount to unstake is more than balance
 # Get pending withdraw amount
✓ Should return pending withdraw amount (576ms)
 # Oracle unstake
✓ Should allow to unstake (51ms)
✓ Should revert if invalid sender
✓ Should revert if there is no currency waiting to unstake
 # Oracle withdraw unstaked
✓ Should allow to withdraw unstaked (85ms)
✓ Should revert if invalid sender
 # Get total pooled WSX
✓ Should return total pooled WSX (540ms)
 # Calculate mint fee
✓ Should return mint fee
 # Calculate reward fee
✓ Should return reward fee
 # Set mint fee
✓ Should set mint fee
✓ Should revert if invalid sender
✓ Should revert if mint fee greater than 35%
 # Set reward fee
✓ Should set reward fee
✓ Should revert if invalid sender
✓ Should revert if reward fee greater than 35%

 # Set DAO address
✓ Should set dao address
✓ Should revert if invalid sender
✓ Should revert if DAO address is zero
 # Set compound stake proxy address
✓ Should set compound stake proxy address
✓ Should revert if invalid sender
✓ Should revert if compound stake proxy address is zero
 # Add oracle
✓ Should add oracle
✓ Should revert if invalid sender
✓ Should revert if trying to add same oracle
 # Add admin
✓ Should add admin
✓ Should revert if invalid sender
✓ Should revert if trying to add same admin
 # Add other roles
✓ Should add other roles
 # Remove oracle
✓ Should remove oracle
✓ Should revert if invalid sender
✓ Should revert if address is not a recognized oracle
✓ Should revert if trying to remove last oracle
 # Remove admin
✓ Should remove admin
✓ Should revert if invalid sender
✓ Should revert if address is not a recognized admin
✓ Should revert if admin trying to revoke himself
 # Remove other roles
✓ Should allow to remove other roles
 # Renounce from the role
✓ Should renounce from the oracle role
✓ Should renounce from the arbitrary role
✓ Should revert when renounce from the admin role
✓ Should revert if trying to renounce another user
✓ Should revert if user does not have the oracle role
✓ Should revert when trying to revoke the last oracle
✓ Should revert if user does not have admin role
✓ Should revert if last admin tries to renounce
 # Oracle claimed rewards
✓ Should allow to claim (516ms)
✓ Should revert if invalid sender

WstWSX

 # Wrap

✓ Should allow to wrap
✓ Should revert if amount to wrap is zero

 # Unwrap

✓ Should allow to unwrap all tokens
✓ Should allow to unwrap half of user's tokens
✓ Should revert if amount to unwrap is zero

 # Claim

✓ Should allow to claim (114ms)
✓ Should revert if there is nothing to claim
✓ Should revert if there is not enough WSX to satisfy claim.
✓ Should revert if Insufficient time passed since unstake. (81ms)

 # Amount getters

✓ Should allow to get amount of wstWSX for a given amount of stWSX
✓ Should allow to get amount of stWSX for a given amount of wstWSX
✓ Should allow to get amount of stWSX for a one wstWSX
✓ Should allow to get amount of wstWSX for a one stWSX

Scenarios

✓ Double conversion of WSX (591ms)
✓ Rounding error from `balanceOf()`
✓ Rounding error for shares transferring (143ms)
✓ Because of rounding errors, different amounts of shares might end in same amount of tokens.
✓ Inflation attack (108ms)
✓ Should return total pooled WSX (540ms)
✓ Pooled approach may end in losses for late users
✓ Scenario for High-3 issue

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% FUNCS	% BRANCH	% FUNCS
IStWSX.sol	100	100	100
StWSX.sol	99.08	92.22	100
WstWSX.sol	100	100	100
Total %	**99.69**	**97.4**	**100**

We are grateful for the opportunity to work with the LiquiStake team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the LiquiStake team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo