Three Sigma

Code Audit

Thunderhead Labs stElx

Disclaimer

Code Audit

Thunderhead Labs stElx

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Code Audit

Thunderhead Labs stElx

Table of Contents

Summary

Code Audit

Thunderhead Labs stElx

Summary

Three Sigma audited Thunderhead Labs's stElx and stHype in a 2.4 person week engagement. The audit was conducted from 11/02/2025 to 18/02/2025.

Protocol Description

Thunderhead is a modular liquid staking protocol enabling networks to deploy Ethereum-grade LST ecosystems with decentralized operator sets, deep liquidity, and instant staking rewards. Supports governance, non-custodial staking, and price aggregation for efficient LST trading.

Scope

Code Audit

Thunderhead Labs stElx

Scope

Filepath	nSLOC
src/Burner.sol	135
src/Delegator.sol	61
src/Minter.sol	39
src/Overseer.sol	270
src/Ownership.sol	14
src/token/stelx.sol	269
src/token/tStorage.sol	22
src/token/wstelx.sol	86
Total	**896**

Assumptions

The OpenZeppelin and Solmate libraries are considered secure.

Methodology

Code Audit

Thunderhead Labs stElx

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit we report our findings using as a guideline Immunefi's vulnerability taxonomy, which can be found at immunefi.com/severity-updated/. The final classification takes into account the severity, according to the previous link, and likelihood of the exploit. The following table summarizes the general expected classification according to severity and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Severity / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Code Audit

Thunderhead Labs stElx

Project Dashboard

Application Summary

Name	stElx
Commit	c8a9fcd
Language	Solidity
Platform	Ethereum

Engagement Summary

Timeline	11/02/2025 to 18/02/2025
№ of Auditors	2
Review Time	6 days

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	0	0	0
High	1	1	0
Medium	1	1	0
Low	6	5	1

None	4	3	1

Category Breakdown

Suggestion	4
Documentation	0
Bug	8
Optimization	0
Good Code Practices	0

Findings

Code Audit

Thunderhead Labs stElx

Findings

3S-stElx-H01

Loss of delegation rewards after unstaking

Id	3S-stElx-H01
Classification	High
Severity	High
Likelihood	High
Category	Bug
Status	Addressed in #c9b3e9c.

Description

The function **`Delegator#unstakePartial`** is responsible for partially unstaking a delegator's funds. It performs multiple operations through **`StakeManager`**, including signaling redelegation, undelegating, signaling unstaking, and finally unstaking the specified amount. However, within **`StakeManager#undelegate`**, the **`delegates[msg.sender]`** value is reset to the zero address, meaning the delegator is no longer assigned to any validator.

The **`Overseer`** contract does not implement any function to call **`Delegator#delegate`**, preventing automatic re-delegation after an unstake operation. As a result, once a delegator unstakes, they will not be delegating to any validator, leading to a permanent loss of delegation rewards.

Recommendation

Modify **`Delegator#unstakePartial`** to call **`stakeManager.delegate(validator)`** after unstaking, ensuring the delegator is immediately reassigned to a validator and continues earning rewards. The corrected implementation should be:

```
function unstakePartial(uint256 amount) public onlyRole(OVERSEER_ROLE) {
    stakeManager.signalRedelegate();
    stakeManager.undelegate();
```

```
    stakeManager.signalUnstake();
    stakeManager.unstake(amount);
    elx.transfer(overseer, amount);
+   stakeManager.delegate(validator);
}
```

This change ensures that delegation remains intact after unstaking, preventing unnecessary reward losses.

3S-stElx-M01

Inflation attack via minting to zero address in **`stElx`** contract

Id	3S-stElx-M01
Classification	Medium
Severity	High
Likelihood	Low
Category	Bug
Status	Addressed in #a832111.

Description

The **`stElx`** contract functions similarly to an ERC-4626 vault, incorporating a virtual offset mechanism that introduces a six-decimal difference between assets and shares. However, it does not implement virtual shares. In the current deployment, **`initialSupply_`** is set to zero, making the contract susceptible to an inflation attack if an attacker can manipulate the price of a share.

This vulnerability arises due to two primary issues. First, the **`Minter#mint`** function does not prevent minting to the zero address. Second, in the **`_transfer`** function of **`stElx`**, if both **`from`** and **`to`** are the zero address, the **`preSyncSupply`** variable increases by **`amount`** due to the placement of the **`if (to == address(0))`** condition inside an **`else`** statement. However, the **`_transferVotingUnits`** function does not increase the total shares in this scenario. Consequently, minting to the zero address inflates the price of a share, enabling an attacker to front-run legitimate transactions and extract value maliciously.

```
function _transfer(address from, address to, uint256 amount) internal {
    uint256 shares = _balanceToShares(amount);
    if (from == address(0)) {
        preSyncSupply += SafeCast.toUint96(amount);
    } else if (to == address(0)) {
        preSyncSupply -= SafeCast.toUint96(amount);
    }
    _transferVotingUnits(from, to, shares);
    emit Transfer(from, to, amount);
}
```

An attacker can exploit this by minting a minimum share before significantly inflating the price of a share. If a user subsequently attempts to mint, the higher price per share may result in the mint transaction yielding zero shares, effectively transferring their assets to the attacker. The exploit enables the attacker to burn their artificially inflated shares and redeem a disproportionate amount of assets, causing loss to the legitimate minter.

Although the virtual offset mitigates the impact to some extent by requiring a large initial capital, the exploit remains viable.

Proof of Concept (PoC):

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;
import "forge-std/Test.sol";
import "./MainMigration.sol";
import "forge-std/console.sol";
contract PoC is MainMigration {
    address alice = makeAddr("alice");
    address bob = makeAddr("bob");
    uint256 amount = 1e18;
    uint256 MINIMUM_MINT_AMOUNT = 1000;
    function setUp() public {
        dealElx(alice, amount);
        dealElx(bob, amount * MINIMUM_MINT_AMOUNT * 1e6 + 1);
        vm.prank(alice);
        elx.approve(address(minter), type(uint256).max);
        vm.prank(bob);
        elx.approve(address(minter), type(uint256).max);
    }
    function test_PoC() public {
        // Bob front-runs Alice's transaction
        vm.startPrank(bob);
        minter.mint(bob, MINIMUM_MINT_AMOUNT);
        minter.mint(address(0), amount * MINIMUM_MINT_AMOUNT * 1e6 -
MINIMUM_MINT_AMOUNT + 1);
        vm.stopPrank();

        vm.prank(alice);
        minter.mint(alice, amount);
        require(stelx.sharesOf(alice) == 0);
    }
```

```
}
```

Recommendation

The contract should explicitly revert transactions where **`Minter#mint`** is called with **`to == address(0)`**.

3S-stElx-L01

Potential revert in **burnAndRedeemIfPossible** due to minimum burn requirement

Id	3S-stElx-L01
Classification	Low
Severity	Low
Likelihood	Low
Category	Bug
Status	Addressed in #2fae1ac.

Description

The function **burnAndRedeemIfPossible** facilitates burning and redemption by attempting to burn the specified **amount** and immediately redeem if possible. If **amount** exceeds **maxRedeemable()**, the function first burns **maxRedeemable** and redeems it before burning the remaining **amount - maxRedeemable**.

However, if **amount - maxRedeemable** is less than **MINIMUM_BURN_AMOUNT**, the second burn operation will revert due to the minimum burn requirement. This causes the entire transaction to revert, not because of insufficient redeemable funds but due to failing the minimum burn constraint. This issue disrupts the expected flow of the function and may create unnecessary transaction failures.

Recommendation

Modify the function to ensure that if **amount - maxRedeemable** is below **MINIMUM_BURN_AMOUNT**, the initial burn amount is adjusted accordingly. This guarantees that the second burn meets the minimum threshold while maintaining the invariant that the total burned amount remains equal to **amount**. The corrected implementation should be:

```
if (amount > maxRedeemable) {
    uint256 firstBurn = maxRedeemable;
    uint256 secondBurn = amount - maxRedeemable;
    // Ensure secondBurn meets the minimum burn requirement
    if (secondBurn < MINIMUM_BURN_AMOUNT) {
```

```
        uint256 diff = MINIMUM_BURN_AMOUNT - secondBurn;
        // If firstBurn falls below MINIMUM_BURN_AMOUNT, revert early to
prevent failure later
        if (maxRedeemable < diff || maxRedeemable - diff <
MINIMUM_BURN_AMOUNT) revert BelowMinimumBurnAmount();
        firstBurn = maxRedeemable - diff;
        secondBurn = MINIMUM_BURN_AMOUNT;
    }
    burnId = burn(to, firstBurn, communityCode);
    redeem(burnId);
    burnId = burn(to, secondBurn, communityCode);
} else {
    burnId = burn(to, amount, communityCode);
    redeem(burnId);
}
```

This adjustment ensures that the second burn operation always satisfies the minimum burn requirement, preventing unnecessary transaction failures while preserving the intended logic of the function.

3S-stElx-L02

Missing event emission in **`unstake(string[])`** function

Id	3S-stElx-L02
Classification	Low
Severity	Low
Likelihood	High
Category	Bug
Status	Addressed in #5a47f39.

Description

The function **`Overseer#unstake(string[] memory offboarding)`** facilitates the unstaking process for multiple delegators. It calculates the total amount to be unstaked using **`pendingUnstake`** and iterates through the provided **`offboarding`** list to unstake the required amounts. However, if **`toUnstake < stake`** for any delegator, the function returns early without emitting the **`Unstaked`** event. This results in an absence of logs for certain unstake operations, making it difficult to track partial unstaking events on-chain.

Recommendation

Before returning in the case where **`toUnstake < stake`**, emit the **`Unstaked`** event to ensure proper event logging. The corrected implementation should be:

```
function unstake(string[] memory offboarding) public onlyRole(MANAGER_ROLE)
{
    uint256 toUnstake = pendingUnstake();
    uint256 totalUnstaked;
    for (uint256 i = 0; i < offboarding.length; i++) {
        uint256 stake =
stakeManager.stakedBalance(address(delegators[offboarding[i]]));
        uint256 necessary = toUnstake > stake ? stake : toUnstake;
        delegators[offboarding[i]].unstakePartial(necessary);
        if (toUnstake < stake) {
            emit Unstaked(i + 1, totalUnstaked + necessary);
            return;
        }
```

```
            toUnstake -= stake;
            totalUnstaked += stake;
        }
        emit Unstaked(offboarding.length, totalUnstaked);
    }
```

This modification ensures that every unstake action, even those that trigger an early return, is properly recorded on-chain.

3S-stElx-L03

Staking failure due to jailed validator prevents cycle completion

Id	3S-stElx-L03
Classification	Low
Severity	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #1359bc6.

Description

The function **`Overseer#stake()`** executes staking for delegators sequentially based on the **`cycleIndex`**, requiring previous delegators to complete their staking allocation before the next one can proceed. The function iterates through the **`operators`** list, staking the required amounts for each delegator via **`Delegator#stake`**, which in turn calls **`StakeManager#stake`**. However, **`StakeManager#stake`** enforces a **`notInJail`** modifier that reverts if the associated validator is jailed.

If a validator is jailed, the staking operation for its associated delegator fails, causing the entire staking cycle to halt. This prevents subsequent delegators from staking until the validator is unjailed, leading to an unintended disruption in the staking process.

Recommendation

Wrap the staking call in a **`try/catch`** block to ensure that if staking fails for a jailed validator, the delegator is skipped rather than halting the entire cycle. Additionally, emit an event when a staking failure occurs to provide on-chain visibility. The corrected implementation should be:

```
function stake() external {
    if (cycleIndex >= operators.length) revert CycleCompleted();
    uint256 toBeStaked = pendingStake();
    uint256 totalStaked;
    uint256 count;
    for (uint256 i = cycleIndex; i < operators.length; i++) {
        if (toBeStaked > cycle[operators[i]]) {
```

```
            try delegators[operators[i]].stake(cycle[operators[i]]) {
                toBeStaked -= cycle[operators[i]];
                count += 1;
                totalStaked += cycle[operators[i]];
                cycle[operators[i]] = 0;
            } catch {
                emit StakingFailed(operators[i], cycle[operators[i]]);
            }
        } else {
            try delegators[operators[i]].stake(toBeStaked) {
                cycle[operators[i]] -= toBeStaked;
                cycleIndex = i;
                count += 1;
                totalStaked += toBeStaked;
                emit Staked(count, totalStaked);
                return;
            } catch {
                emit StakingFailed(operators[i], toBeStaked);
            }
        }
    }
    emit Staked(count, totalStaked);
    cycleIndex = operators.length;
}
```

This modification ensures that a jailed validator does not block the entire staking process.

3S-stElx-L04

Incorrect event emission in **`stake`** due to premature zeroing of **`cycle`** values

Id	3S-stElx-L04
Classification	Low
Severity	Low
Likelihood	High
Category	Bug
Status	Addressed in #3cf64f1.

Description

In the **`Overseer`** contract, the **`stake`** function moves pending stakes to delegators, who then stake the funds with their associated validators.

```
    function stake() external {
        if (cycleIndex >= operators.length) revert CycleCompleted();
        uint256 toBeStaked = pendingStake();
        uint256 totalStaked;
        uint256 count;
        for (uint256 i = cycleIndex; i < operators.length; i++) {
            if (toBeStaked > cycle[operators[i]]) {
                delegators[operators[i]].stake(cycle[operators[i]]);
                toBeStaked -= cycle[operators[i]];
1>              cycle[operators[i]] = 0;
                count += 1;
2>              totalStaked += cycle[operators[i]];
            } else {
                delegators[operators[i]].stake(toBeStaked);
                cycle[operators[i]] -= toBeStaked;
                cycleIndex = i;
                count += 1;
                totalStaked += toBeStaked;
                emit Staked(count, totalStaked);
                return;
            }
        }
        emit Staked(count, totalStaked);
```

```
        cycleIndex = operators.length;
    }
```

The issue occurs when **`toBeStaked > cycle[operators[i]]`**. In this case:

- The function sets **`cycle[operators[i]] = 0`** (**1›**) before adding its value to **`totalStaked`** (**2›**).

- Since **`cycle[operators[i]]`** is already zeroed out, the addition contributes **`0`**, leading to an incorrect value for **`totalStaked`**.

- As a result, the **`Staked`** event emits an incorrect total stake amount.

Recommendation

Zero out **`cycle[operators[i]]`** **after** adding it to **`totalStaked`**:

```
            if (toBeStaked > cycle[operators[i]]) {
                delegators[operators[i]].stake(cycle[operators[i]]);
                toBeStaked -= cycle[operators[i]];
                count += 1;
                totalStaked += cycle[operators[i]];
+               cycle[operators[i]] = 0;
            } else {
```

This ensures that **`totalStaked`** correctly reflects the amount being staked before resetting the storage value.

3S-stElx-L05

Rounding in **`_transfer`** favors caller on burning, leading to underestimated burned shares

Id	3S-stElx-L05
Classification	Low
Severity	Low
Likelihood	High
Category	Bug
Status	Acknowledged

Description

In tokenized protocols, rounding should generally favor the protocol rather than the caller to prevent potential exploitation. However, in the **`stelx#_transfer`** function, the conversion of balance to **`shares`** is always rounded down using **`_balanceToShares`**.

This behavior creates an issue during burns:

- When tokens are burned, the corresponding **`shares`** are rounded down, meaning fewer shares are deducted than expected.

- This results in an underestimation of burned shares, benefiting the caller at the protocol's expense.

Recommendation

- Modify **`_balanceToShares`** to support both rounding up and rounding down.

- In **`_transfer`**:

  - Round **`shares up`** when burning.

  - Round **`shares down`** when minting.

3S-stElx-L06

Unbounded loop in **getBurnIds** and **getBurns** can cause view function failure

Id	3S-stElx-L06
Classification	Low
Severity	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #ce657e1.

Description

In the **Burner** contract, the functions **getBurnIds** and **getBurns** are used to retrieve burn information for a given address. These functions rely on an internal function, **_getBurnIds**, which iterates over an unbounded storage array, **burns**.

Although **getBurnIds** and **getBurns** are **view** functions, they are still subject to gas limits. If the **burns** array grows too large, the gas cost of iterating through it may exceed the limit, causing these functions to fail. This can make it impossible to retrieve burn history on-chain for large datasets.

Recommendation

- Introduce a mapping to track burn IDs per address:

```
mapping(address => uint256[]) accountToIds;
```

- In the **burn** function, append new burn IDs to this mapping:

```
accountToIds[msg.sender].push(burns.length - 1);
```

- Modify **_getBurnIds** to return the pre-tracked burn IDs instead of iterating over **burns**:

```
return accountToIds[account];
```

3S-stElx-N01

Lack of input validation in setter functions

Id	3S-stElx-N01
Classification	None
Category	Suggestion
Status	Addressed in #892be56.

Description

The setter functions **`changeEpochLength`**, **`changeRewardShare`**, **`setDelegatorImplementation`**, **`setProxyFactory`**, **`setAprThresholdBps`**, and **`setSlashThresholdBps`** in the **`Overseer`** contract are designed to update critical parameters and addresses within the contract. However, they currently lack input validation, which could lead to potential issues such as setting parameters to inappropriate values or assigning critical addresses to the zero address due to human error.

Recommendation

Implement input validation checks within each setter function to ensure that the provided values are within acceptable bounds. For address parameters, ensure that they are not set to the zero address. For numerical parameters, it would be beneficial to get them bounded to reasonable limits to prevent setting values that could disrupt the contract's intended functionality.

3S-stElx-N02

Lack of SafeERC20 usage for ELX token transfers

Id	3S-stElx-N02
Classification	None
Category	Suggestion
Status	Acknowledged

Description

In the provided smart contracts, including **`Burner`**, **`Delegator`**, **`Minter`**, **`Overseer`**, and **`wstelx`**, the **`ELX`** token transfers are not utilizing the SafeERC20 library. The SafeERC20 library is a well-established standard for handling ERC20 token transfers safely, ensuring that all token transfer operations are executed correctly and revert in case of failure. Although the **`ELX`** token source code shared by the client does not require the use of SafeERC20, it is prudent to incorporate this library as a precautionary measure since the token is not yet deployed.

Recommendation

It is recommended to integrate the SafeERC20 library for all **`ELX`** token transfer operations within the **`Burner`**, **`Delegator`**, **`Minter`**, **`Overseer`**, and **`wstelx`** contracts.

3S-stElx-N03

Zero amount in stake and unstake operation continues loop execution

Id	3S-stElx-N03
Classification	None
Category	Suggestion
Status	Addressed in #c9debae.

Description

The **`stake()`** and **`unstake()`** functions in the **`Overseer`** contract are responsible for distributing pending stakes and managing unstaking across multiple operators according to their cycle allocations. When **`toBeStaked`** or **`toUnstake`** become zero during the loop execution, the function continues to process one further loop unnecessarily, potentially leading to zero-amount stake and unstake operations and wasteful gas consumption. The functions' purposes are to distribute available stake amounts proportionally across operators based on their cycle allocations and manage unstaking efficiently, but the current implementations do not optimize for the case when all available stake or unstake amounts have been distributed.

Recommendation

Add a check at the beginning of the loop to break execution when **`toBeStaked`** reaches zero, and refactor the check in **`unstake()`** to **`if (toUnstake <= stake) return`**.

`Overseer#stake`

```
function stake() external {
    // ... existing code ...
    for (uint256 i = cycleIndex; i < operators.length; i++) {
+       if (toBeStaked == 0) break;

        if (toBeStaked > cycle[operators[i]]) {
            // ... existing code ...
        } else {
            // ... existing code ...
        }
```

```
    }
    // ... existing code ...
}
```

Overseer#unstake

```
function unstake(string[] memory offboarding) public onlyRole(MANAGER_ROLE)
{
    uint256 toUnstake = pendingUnstake();
    uint256 totalUnstaked;
    for (uint256 i = 0; i < offboarding.length; i++) {
        uint256 stake =
stakeManager.stakedBalance(address(delegators[offboarding[i]]));
        uint256 necessary = toUnstake > stake ? stake : toUnstake;
        delegators[offboarding[i]].unstakePartial(necessary);
-       if (toUnstake < stake) return;
+       if (toUnstake <= stake) return;
        toUnstake -= stake;
        totalUnstaked += stake;
    }
    emit Unstaked(offboarding.length, totalUnstaked);
}
```

3S-stElx-N04

Overwriting **`delegators[name]`** can lead to loss of access to delegator contracts

Id	3S-stElx-N04
Classification	None
Category	Suggestion
Status	Addressed in #619dd4a.

Description

The **`Overseer#addOperator`** function allows an administrator to register a new operator by deploying a **`Delegator`** contract and storing its address in the **`delegators`** mapping under the given **`name`**. However, if **`delegators[name]`** already contains an address, the new assignment will overwrite the existing value, making the previously deployed **`Delegator`** contract inaccessible. This could result in the loss of access to staked funds within the overwritten **`Delegator`** contract.

Although the function is restricted to the administrator, adding a defensive check to prevent name collisions would mitigate the risk of accidental overwrites.

Recommendation

Before assigning a new **`Delegator`** contract to **`delegators[name]`**, the function should check whether an entry already exists and revert if it does. Modify the function as follows:

```
require(delegators[name] == address(0), "Operator name already in use");
```

This ensures that each operator name is unique and prevents accidental loss of access to previously deployed **`Delegator`** contracts.