Element369

Security review

Version 1.0

Reviewed by
nmirchev8
ge6a
deth

Table of Contents

1 About Egis Security

Egis Security is a team of experienced smart contract researchers, who strive to provide the best smart contract security services possible to DeFi protocols.

The team has a proven track record on public auditing platforms like Code4rena, Sherlock, and Cantina, earning top placements and rewards exceeding $170,000. They have identified over 150 high and medium-severity vulnerabilities in both public contests and private audits.

2 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	Element369
Repository	Private
Commit hash	c312495072c48b6835999692aff9d47abc3c6b2b
Resolution	055e20d5c1cf22287691a85d8a892f15749e072a
Documentation	https://docs.helios-hlx.win/element-369
Methods	Manual review

Scope

contracts/DevDistribute.sol
contracts/Element369HolderVault.sol
contracts/Element369NFT.sol
contracts/ElementBuyBurnV2.sol
contracts/FluxHub.sol
libs/*

Issues Found

Critical risk	0
High risk	2
Medium risk	2
Low risk	2
Informational	1

5 Findings

5.1 High

5.1.1 Lock of funds from the 777 reward pool

Severity: *High risk*

Context: Element369HolderVault.sol#L111

Description: In the `updateCycle()` function, when the `startCycleId` for the corresponding 777 cycle begins, the total number of multipliers at that moment is saved in `cycles777[currentCycle777].multiplierPool`. When a user who owns an NFT claims the 777 reward, `_processTokenIdCycles777` is called, where it checks if the NFT was minted after the `cycle.startCycleId` for that cycle and if it was burned before `cycle.endCycleId`. The `endCycleId` is set in `updateCycle()` and represents the 11-day cycle number during which the next 777 cycle begins.

The issue is that if an NFT is minted after the start date of the corresponding cycle but is burned before the cycle ends, `cycles777[currentCycle777].multiplierPool` would not be reduced. This means that the respective share of the 777 reward pool cannot be withdrawn by the user or anyone else and will remain locked in the contract.

Recommendation: The solution is, instead of saving the `multiplierPool` at the beginning of the 777 cycle, to save it at the end, as is done for the other rewards.

Resolution: Fixed

5.1.2 Stealing staking rewards for other users

Severity: *High risk*

Context: Element369HolderVault.sol#L123x

Description: The FluxHub contract stakes a portion of the received Flux tokens in the Flux staking contract, where rewards are distributed every 8, 28, 88, and 777 days. The tokens given as rewards are received in the contract through the `distribute()` function and are divided among the pools for 8, 28, 88, and 777 days in the ratio 38%, 30%, 22%, and 10%, respectively. Before calling the main actions in the contract, stake and claim, the `updateRewardsIfNecessary()` function is called, which checks if the day for releasing the accumulated rewards has arrived, and if so, it updates the `rewardPerShare` variable. When a user claims, they receive rewards for the difference between `rewardPerShare` and the value recorded in `rewardDebt` for their NFT. In other words, when a user stakes tokens, they won't receive rewards for the time before that, accumulated by other users.

The issue is that the above description is not valid for the Element369 protocol. A user can front-run `updateCycle()`, mint an NFT at the end of the cycle, and receive rewards from it even though they did not stake during the cycle, and their deposit wasn't used for accumulating staking rewards. In other words, by doing this, one can steal from the rewards of users who have been staking continuously. For example, if a user opens a position just before the end of the 88-day cycle, they could take a large share of the 22% of rewards accumulated over those 88 days by regular users, even though they haven't contributed a single day of staking. This results in a loss of funds for other users. This problem could be exacerbated if, for some reason, the `distributeTreasury()` rewards function hasn't been called for a long time, resulting in a significant accumulation of undistributed rewards.

Recommendation: A possible and easily implementable solution would be to introduce a cooldown period, during which new users cannot receive staking rewards until it expire.

Resolution: Fixed

5.2 Medium risk

5.2.1 `cycle.endCycleId` bypass in `_processTokenIdCycles777()`

Severity: *Medium risk*

Context: FluxHub.sol#L212

Description: The function `FluxHub.distribute777Treasury` is used to send accumulated rewards in the 777 treasury to `Element369HolderVault`. There, the `Element369HolderVault.register777CycleTokens` function is called to account for the sent rewards, which can then be claimed by users. When a user claims rewards, it is verified whether the corresponding NFT is eligible for these rewards, meaning it was created before the start of the respective 777 cycle and was not burned before its end. The end of a 777 cycle is not predetermined; it is set by the `Element369HolderVault.updateCycle` function when the next cycle begins.

If, for any reason, `distribute777Treasury` is executed before `updateCycle()` has set the end (as the 11-day reward cycle number) of the 777 cycle, a user who burned their NFT before the end of the period could still receive rewards from this cycle because `cycles777[cycles777Available].endCycleId` would be 0.

It is important to consider when this could happen in practice. The `distribute777Treasury` function can only be called by whitelisted users. However, they may decide it is necessary to execute the function earlier due to more favorable market conditions for swap operations. It is also possible that the 777 cycle has effectively ended, but the `updateCycle()` function has not yet been executed. Notably, the `FluxHub` contract does not track 11-day cycles but only checks the remainder when divided by 777 to determine the current 777 cycle. The `updateCycle()` function might not execute in time due to several reasons:

1) no one wants to execute it
2) there may not be enough accumulated rewards, leading to a revert (a minimum amount of each token must be available)
3) the 11-day period may be ongoing; for example, the 777 cycle change occurs right after the function is already called, requiring a wait of almost 11 days.

Recommendation: Two possible solutions: - to add an additional check in _processTokenIdCycles777() to prevent claiming rewards if endCycleId is 0. - to set endCycleId in advance

Resolution: Fixed

5.2.2 `updateCycle()` DoS and delays

Severity: *Medium risk*

Context: Element369HolderVault.sol#L101

Description: The `updateCycle()` function creates a new 11-day cycle and determines the reward amount per multiplier unit. For this function to execute, the following conditions must be met:

1) At least 11 days must have passed since the last successful execution.
2) Each of the three tokens - flux, e280, and inferno must have at least `minCyclePool` amount available for rewards; otherwise, `_getNextCyclePool()` would revert.

The tokens e280 and inferno are acquired as `FluxHub` swaps titanx for them on Uniswap. Flux is earned from the Flux auction by using the amount of titanx with which users purchased NFTs. In the `Element369NFT` contract, the functions through which NFTs are purchased have a `onlySale` modifier that checks if there is an active sale period. Sale periods last 44 days and alternate with 44-day periods when NFTs cannot be purchased. This means that during these 44-day periods, the flow of Flux to the Vault will be interrupted. A malicious user could exploit this by claiming all available flux from the auction, executing `updateCycle()` with it at the beginning of the period when NFTs cannot be purchased, thus blocking rewards claiming and the creation of new cycles for 44 days. If there is insufficient availability of the other two tokens, a donation can be made to carry out the attack. If a user burns their NFT on the 30th day, they will receive no rewards for the cycle, as their `burnCycle` will match the current cycle due to the inability to execute `burnCycle`.

In the following lines, I will describe another impact, which is largely related to the above description but with some nuances. The start of each 777 reward cycle is set in the `Element369HolderVault` constructor as the number of the 11-day cycle:

First 777 cycle, 35th 11-day cycle, day 35 * 11 = 385
Second 777 cycle, 105th 11-day cycle, day 105 * 11 = 1155
Third 777 cycle, 175th 11-day cycle, day 175 * 11 = 1925
Fourth 777 cycle, 245th 11-day cycle, day 245 * 11 = 2695

The issue with this time measurement method is that the 11-day cycles depend on the execution of the `updateCycle()` function, whereas the 777-day cycles depend only on the elapsed days. Therefore, if `updateCycle()` is not executed, the cycle will not advance even if all conditions are met. Additionally, if the function has not been executed for an extended period, say 22 days, it will increase the 11-day cycle number by only 1, not 2. This discrepancy in calculating the 11-day cycles can lead to a significant delay relative to the 777-day cycles. Over time, this delay can exceed 777 days, which may result in a 777 cycle being skipped in `updateCycle()`, meaning that the skipped cycle will remain with `multiplierPool = 0`, and rewards for it cannot be distributed due to division by zero. Users who have maintained their positions for the entire period will have to wait until the next 777 cycle to receive their rewards, which will, however, be shared with new participants joining the new cycle.

It's worth noting that delays, besides being caused by the attack described above, may also accumulate naturally if no one has the incentive to execute the `distributeReward` function or the `updateCycle()` function over certain periods. Additionally, delays may occur if enough rewards are not generated.

Recommendation: Use time instead of manual cycle creations

Resolution: Fixed

5.3 Low risk

5.3.1 Ineffective slippage protection in `buyAndBurn()`

Severity: *Low risk*

Context: SwapActions.sol#L156

Description: The `ElementBuyBurnV2.buyAndBurn()` function is used to buy Element 280 tokens using TitanX or WETH and then burn them. The parameters `minE280Amount` and `minTitanXAmount` are provided, indicating the minimum amount of each token that should be received from each swap. This serves as a form of slippage protection. The problem is that the user calling the function cannot know the available amount of each token when the transaction is executed, as another user could make a deposit, increasing the balances. A malicious actor could exploit this by performing a sandwich attack on the swap operation, causing high slippage and resulting in a loss for the protocol and a profit for themselves. The standard solution in such situations is for the user to use a contract to execute all operations atomically in one transaction. However, in this case, this is not possible because there is a check in buyAndBurn that prevents the function from being called by a contract.

NOTE Ether swap cap is initially set to 2 ether, which is big value, which may be subject to sandwich attacks.

Recommendation: One option is to make the transfer of TitanX and WETH occur within `buyAndBurn` so that the operation is atomic. Another option is to add an amount parameter to the function.

Resolution: Acknowledged

5.3.2 No `unstake()` feature in `FluxHub`

Severity: *Low risk*

Context: FluxHub.sol#L25

Description:

In the FluxHub contract, staking Flux in the Flux staking contract to receive staking rewards has not been implemented. The issue is that the reverse function for unstaking the staked Flux is also not implemented. I understand that, according to the protocol's design, unstaking this amount is not intended under normal circumstances. However, without this functionality, in case of an emergency, there would be no way to withdraw the tokens and restore the protocol, which is also not upgradable.

Recommendation It is recommended that an unstacking functionality be implemented that can only be executed by the owner as a safeguard in case of unforeseen circumstances.

Resolution: Acknowledged

5.4 Informational

5.4.1 `enterAuction()` donation attack

Severity: *Informational*

Context: Auction.sol#L311

Description The Flux auction emits a certain amount of Flux each day, with the supply decreasing by a set percentage each week. The emitted amount is distributed proportionally to the amount of titanx deposited by each user. If a user participates in the auction for a given day and sees in the mempool that the protocol has purchased NFTs, they can prevent the protocol's participation in the auction for that day by front-running the transaction, depositing a minimal amount of titanx, and executing FluxHub.enterAuction(). This way, enterAuction() will be blocked for one day, and the protocol will not be able to participate in the auction, allowing the user to claim a larger share of the emitted tokens for themselves.

Recommendation A possible solution is to implement a min amount requirement in `enterAuction()`

Resolution: Fixed