marchev

Element 369 Security Review

Version 1.1

Reviewed by:

Martin Marchev

October 30, 2024

Table of Contents

1 Introduction 3
1.1 About Martin Marchev 3
1.2 Disclaimer 3
1.3 Risk Classification 3
2 Executive Summary 4
2.1 About Element 369 4
2.2 Synopsis 4
2.3 Issues Found 4
3 Findings 5

1 Introduction

1.1 About Martin Marchev

Martin Marchev is an independent security researcher specializing in web3 security. His track record includes top placements in competitive audits such as 1st place in the Immunefi Arbitration contest, 2nd place in the PartyDAO contest (as a team), 3x Top 10 and 5x Top 25 rankings. Martin has responsibly disclosed vulnerabilities in live protocols on Immunefi and has been involved in high-profile audits of projects exceeding $1B in Total Value Locked (TVL), demonstrating his ability to navigate and address complex security challenges. For bookings and security review inquiries, you can reach out on Telegram: https://t.me/martinmarchev

1.2 Disclaimer

While striving to deliver the highest quality web3 security services, it is important to understand that the complete security of any protocol cannot be guaranteed. The nature of web3 technologies is such that new vulnerabilities and threats may emerge over time. Consequently, no warranties, expressed or implied, are provided regarding the absolute security of the protocols reviewed. Clients are encouraged to maintain ongoing security practices and monitoring to address potential risks.

1.3 Risk Classification

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

1.3.1 Severity Criteria

- **Critical -** Severe loss of funds, complete compromise of project availability, or significant violation of protocol invariants.
- **High -** High impact on funds, major disruption to project functionality or substantial violation of protocol invariants.
- **Medium -** Moderate risk affecting a noticeable portion of funds, project availability or partial violation of protocol invariants.
- **Low -** Low impact on a small portion of funds (e.g. dust amount), minor disruptions to service or minor violation of protocol invariants.
- **Informational -** Negligible risk with no direct impact on funds, availability or protocol invariants.

2 Executive Summary

2.1 About Element 369

Element 369 is an expansion pack NFT collection within the Element 280 ecosystem, allowing holders to passively earn rewards in E280, Inferno and Flux tokens. By participating in perpetual Flux auctions, Element 369 enhances token growth and ecosystem stability, with features like auto-compounding rewards and token-backed value that is transferable upon resale.

2.2 Synopsis

Project Name	Element 369
Project Type	NFT
Repository	Element-369
Commit Hash	bb7e2d5c...6bdc0176
Review Period	21 October 2024 - 29 October 2024

2.3 Issues Found

Severity	**Count**
Critical Risk	0
High Risk	7
Medium Risk	7
Low Risk	1
Informational	5
Total Issues	**20**

3 Findings

High Risk

[H-1] Incorrect multiplier pool accounting leads to reward dilution and irrecoverably stuck funds

Context: Element369NFT.sol#L144-L152

Description: Due to improper accounting of the `multiplierPool` variable in `Element369NFT`, the contract has a vulnerability where user rewards are gradually diluted over time and a portion of the `Element369HolderVault` funds remains permanently trapped in the contract.

Rewards in `Element369HolderVault` are calculated using the formula:

```
share = amount / (totalMultipliers * 2);
```

where `totalMultipliers` is the `multiplierPool` value in `Element369NFT`. Because `multiplierPool` is only increased and not decreased, the share calculation becomes increasingly inaccurate over time, causing user rewards to be diluted as more NFTs are burned. Consequently, funds corresponding to the multipliers of all burned NFTs remain irrecoverably trapped in the `Element369HolderVault` contract.

Recommendation: Update the `Element369NFT` contract to decrease the value of `multiplierPool` whenever NFTs are burned, ensuring accurate reward calculations and no loss of funds.

Resolution: Fixed in d1dab29

[H-2] HLX/ELMNT swap during treasury distribution is susceptible to self-sandwich attacks

Context: FluxHub.sol#L369-L381

Description: The `distributeTreasury()` and `distribute777Treasury()` functions in `FluxHub` facilitate the swapping of TITANX tokens into different assets for treasury distribution. Among these operations, a HLX/ELMNT swap is performed after portion of the TITANX funds are converted to HLX. The HLX/ELMNT swap is executed through a UniswapV2 pool, which lacks the TWAP checks that protect the UniswapV3 swaps in the protocol. This omission exposes the protocol to a *self-sandwich attack*. A malicious actor can exploit this vulnerability by invoking the functions with `0` set as the `e280MinAmountOut` argument, enabling them to self-sandwich their transaction and drain value from the protocol. Due to the low liquidity (~$52K) in the HLX/ELMNT UniswapV2 pool, this type of attack does not require substantial capital. Unlike the UniswapV3 swaps in the protocol, where a TWAP check exists to protect against such attacks, the absence of this protection for the HLX/ELMNT swap makes it vulnerable.

Recommendation: Apply TWAP protection for the HLX/ELMNT swap to prevent self-sandwiching attacks by malicious actors.

Resolution: Fixed in 24bec06 and b79688d

[H-3] `distributionLimit` does not provide sufficient protection against MEV attacks

Context: FluxHub.sol#L171, FluxHub.sol#L195

Description: The `distributeTreasury()` and `distribute777Treasury()` functions in `FluxHub` use a configured `distributionLimit` to mitigate and disincentivize price manipulation MEV attacks. However, the protocol lacks a mechanism to limit how often the `distributeTreasury()` and `distribute777Treasury()` functions can be invoked. This creates an opportunity for an MEV searcher for repeatedly calling these functions in a single block. The attacker could then bundle and sandwich these transactions, participate in an MEV auction and bribe MEV builders to prioritize their transactions in a favorable order. This will effectively bypass the intended protection offered by the `distributionLimit` and create lucrative opportunity for a price manipulation attack. The impact of such attack would be somewhat limited due to the presence of TWAP checks but the attacker could still drain value from the protocol especially if hight amount of TITANX is accumulated.

Recommendation: Introduce a cooldown period or limit the frequency of function calls to prevent sandwich attacks of multiple bundled transactions.

Resolution: Fixed in 24bec06 and b79688d

[H-4] Malicious actor can permanently DoS `FluxHub.startStake()`

Context: FluxHub.sol#L151

Description: The `startStake()` function in `FluxHub` is provided as an alternative to the `claimAuctions()` function to enable users to stake Flux reward payouts when the contract has no claimable Flux auctions and no TitanX balance. The function can only be called once per 24 hours. To ensure these constraints, the function contains the following check:

```
(uint256 titanAmount, uint32 eligibleDays) = _getAvailableTitanX();
if (titanAmount > 0 || eligibleDays == 0) revert Prohibited();
```

As per the `_getAvailableTitanX()` function, the `titanAmount` is calculated as follows:

```
uint256 availableBalance = IERC20(TITANX).balanceOf(address(this)) - treasuryBalance
 ↪  - treasury777Balance; //@audit IERC20(TITANX).balanceOf(address(this)) is
 ↪  manipulable
```

Malicious actor can DoS the function by sending `1 wei` of TITANX to the `FluxHub` contract. Theoretically, they could do that indefinitely by frontrunning the calls to `startStake()` or simply sending `1 wei` to the contract whenever the `FluxHub` TITANX becomes `0`. This will lead to permanent DoS of the `startStake()` function.

A temporary workaround until Flux auctions are still live would be for a user to send TitanX to the contract and call `enterAuction()`. This will effectively stake the available Flux balance in `FluxStaking`. But once Flux auctions are no longer live, this

Recommendation: Remove the requirement for a `0` balance of TITANX in the `startStake()` function. Since `claimAuctions()` already utilizes the entire claimed TITANX balance, the `0` balance check appears unnecessary and can be safely removed to prevent DoS attacks of `startStake()`.

Resolution: Fixed in 4afe4dc

[H-5] NFT holders unable to claim 777 rewards and backing due to incorrect cycle processing logic

Context: Element369HolderVault.sol#L350, Element369HolderVault.sol#L448, Element369HolderVault.sol#L469

Description: Users who mint NFTs after the start of the first 777 cycle are unable to claim 777 rewards and backing for subsequent eligible cycles due to a faulty logic in the `_processTokenIdCycles777()` and `_processTokenIdBacking777()` functions.

Example scenario:

1. Alice mints an NFT during cycle 100.
2. Alice holds her NFT for two full 777-day cycles, totaling 1554 days.
3. Alice attempts to claim her 777 rewards using `claim777Rewards()`.

The expected outcome is that Alice should receive rewards for cycles `cycles777[1]` and `cycles777[2]`. However, due to a bug in the `_processTokenIdCycles777()` function, she is unable to claim any rewards. The issue stems from the following logic:

```
for (uint256 i = 0; i < cycles777Available; i++) {
    Cycle777 memory cycle = cycles777[i];
    if (nftMintCycle > cycle.startCycleId || lastCycle <= cycle.endCycleId) break;
     ↪  //@audit Prevents users who minted after the first cycle start to claim
     ↪  rewards
    infernoPool += cycle.infernoPerMulitplier;
    e280Pool += cycle.e280PerMultiplier;
}
```

The loop terminates prematurely if the NFT was minted after `cycles777[0].startCycleId`. This condition causes the `for` loop to halt, preventing any subsequent 777 cycles from being processed. This is likely due to a mistaken use of `break` instead of `continue`, intended to skip the current cycle and proceed to the next one.

The same issue affects `claim777Rewards()`, `claim777Backing()` and `get777Rewards()`.

Recommendation: Replace the `break;` statement with `continue;` to ensure the loop properly checks all 777 cycles, allowing users to claim rewards for all eligible periods.

Resolution: Fixed in a947473 and 90cbb3a

[H-6] Incomplete support for four 777-day reward cycles causes rewards loss

Context: Element369HolderVault.sol#L91-L93

Description: The `Element369HolderVault` contract is designed to support three 777-day reward cycles:

```
cycles777[0].startCycleId = 35;
cycles777[1].startCycleId = 105;
cycles777[2].startCycleId = 175;
```

However, this reward distribution model does not fully align with the Flux protocol from which the rewards are sourced. The `FluxAuction` which is used to fill up the 777-day treasury runs for 2922 days:

```
function deposit(uint192 _amount) external {
    if (startTimestamp > Time.blockTs()) revert FluxAuction__NotStartedYet();
    if (startTimestamp + 2922 days < Time.blockTs()) {
        revert FluxAuction__AuctionIsOver();
    }
    // ...
}
```

Rewards continue to be distributed to the 777 treasury up to the last day. As a result, rewards accumulated from day 2331 (end of the third 777-day cycle) until day 2922 will be claimable after day 3108 (end of the fourth cycle).

Since the `Element369HolderVault` contract only supports three 777-day cycles, any rewards accrued during this fourth cycle will be inaccessible, leading to a potential loss of rewards and divergence from the Flux protocol's 777-day reward distribution.

Note: This issue cannot currently be reproduced due to the presence of **H-4**, which masks the effects of this issue. However, fixing **H-4** will not automatically resolve this issue. Once **H-4** is addressed, this vulnerability will become reproducible and continue to affect the system.

Recommendation: To align fully with the Flux protocol's 777-day reward distribution scheme, modify the `Element369HolderVault` contract to support four 777-day reward cycles instead of three.

Resolution: Fixed in 432bb1f and 90cbb3a

[H-7] Manual cycle update requirement risks user fund loss when claiming NFT backing

Context: Element369NFT.sol#L150

Description: The current design requires manual interaction to update the reward cycle every 11 days by calling the `updateCycle()` function:

```
function updateCycle() external {
    // ...
    (uint32 cycleId, uint256 totalMultipliers) = nft.updateCycle();
    // ...
    emit CycleUpdated();
}
```

This manual dependency means that, if no user calls `updateCycle()` after a cycle has ended, the protocol remains in the previous cycle. This behavior may lead to potential issues in NFT backing claims. This manual requirement poses a risk as users may not reliably trigger the update at the correct interval, especially as protocol activity fluctuates over time.

For instance, if 12 days pass without an `updateCycle()` call, the protocol does not recognize that a new cycle has begun. Users may then burn their NFTs assuming the cycle has advanced, only to find they are ineligible for backing in the last cycle because the protocol remains in the prior cycle:

```
function batchBurn(uint256[] calldata tokenIds) external {
    if (tokenIds.length == 0) revert ZeroInput();
    for (uint256 i = 0; i < tokenIds.length; i++) {
        uint256 tokenId = tokenIds[i];
        if (ownerOf(tokenId) != msg.sender) revert Unauthorized();
        _burn(tokenId);
        _updateBurnCycle(tokenId, currentCycle, msg.sender); //@audit currentCycle
         ↪  will still be the old cycle if updateCycle() was not called manually
    }
}
```

This behavior can result in inadvertent financial loss for users due to an outdated cycle state.

Example scenario:

1. The protocol starts and Alice mints an NFT on day 1.
2. 12 days pass, but no user calls `updateCycle()`.
3. Alice, assuming she has held her NFT through the first reward cycle (11 days), burns her NFT on day 12 expecting backing.
4. However, since `updateCycle()` was not called, the protocol still considers it to be cycle 1, making Alice ineligible for any backing accumulated during cycle 1.

This issue risks user fund loss due to misalignment between the expected and actual protocol state. Users could unintentionally forfeit backing if the cycle state is not updated as expected.

Recommendation: Implement automatic `updateCycle()` updates whenever user operations are executed that depend on the current cycle state, such as NFT burns, if more than 11 days have elapsed since the last update. This ensures that the protocol reflects the current cycle state without relying solely on manual intervention. Retain the ability for users to manually call `updateCycle()` as a fallback, but automatic updates should be the primary mechanism to maintain accurate cycle accounting and prevent user losses.

Resolution: Acknowledged

Medium Risk

[M-1] Malicious actor can grief the treasury distribution

Context: FluxHub.sol#L360

Description: The treasury distribution functions in `FluxHub` use the `_processTreasurySwaps()` function internally to perform various swaps before distributing the treasury as per the project tokeonomics. One of the performed swaps is TITANX/HLX:

```
_swapUniswapV3Pool(HELIOS, heliosSwapAmount, heliosMinAmountOut, deadline);
```

Before the swap is executed against the HLX/TITANX UniswapV3 pool, a TWAP check is performed against the same pool as a price manipulation protection. The `_twapCheck()` function uses Uniswap's `OracleLibrary` to query `arithmeticMeanTick` that occurred `_secondsAgo` or `oldestObservation` seconds ago (whichever is more recent):

```
uint32 oldestObservation = OracleLibrary.getOldestObservationSecondsAgo(poolAddress);
if (oldestObservation < _secondsAgo) {
    _secondsAgo = oldestObservation;
}

(int24 arithmeticMeanTick,) = OracleLibrary.consult(poolAddress, _secondsAgo);
```

While this is a standard approach to get a UniswapV3 TWAP price for a given time interval, the problem here is that the HLX/TITANX pool uses the default `observationCardinality` of `1`. This basically means that the pool keeps only one historical observation. In UniswapV3, observations are written retroactively at the beginning of the block if there was a price change. These observations are using the current block timestamp as their `blockTimestamp` field. The `OracleLibrary.consult()` function that is used has the following precondition though:

```
require(secondsAgo != 0, 'BP');
```

This means that a malicious actor can grief the treasury distribution mechanism by frontrunning the transaction and performing a swap against the HLX/TITANX pool. This will overwrite the only observation in the pool with a timestamp of `block.timestamp` which will cause the user's transaction to revert due to the check in the `OracleLibrary.consult()` function.

Recommendation: To mitigate this issue, increase the `observationCardinality` of all UniswapV3 pools for which TWAP check is performed via `IUniswapV3Pool.increaseObservationCardinality` ⌋ `Next()`. Ideally, the `observationCardinality` should be sufficiently high to ensure a robust TWAP oracle is available.

Resolution: Fixed via 0x288b...193b

[M-2] Duplicate team wallet configuration in `DevDistribute` will cause funds to be forever stuck in the contract

Context: DevDistribute.sol#L32

Description: The constructor of the `DevDistribute` contract accepts a set of 3 team wallets and the distribution ratio between the wallets. However, the constructor does not check the team wallets for uniqueness as the result of the addition of the wallet to the `_teamWallets` set is not checked:

```
_teamWallets.add(wallet); //@audit Set addition result is not checked
```

So in case of an incorrect configuration which contains duplicate team wallets, this will cause the `shares` for the duplicate wallet to be overwritten:

```
shares[wallet] = share;
```

This will result in the total distribution ratio for the wallets to be less than 100%. The `totalPercents` check in the constructor also does not prevent this issue form happening.

In case of such misconfiguration, the contract will be in a state that cannot be fixed. Using `changeWallet()` cannot solve the issue because it may simply be used to replace any of the already added wallets.

The following example scenario illustrates the issue:

The `DevDistribute` contract is constructed with the following incorrect configuration:

```
_wallets = [wallet1, wallet2, wallet2]
_shares = [30, 40, 30]
```

As a result, the contract is constructed with only 2 wallets - `wallet1` and `wallet2`. The total number of shares is 60 - the second addition of `wallet2` overwrites the 40 shares in the configuration. This leads to only 60% of the funds being distributed upon a `distributeToken()` call. The issue could be mitigated to a certain degree by calling `distributeToken()` multiple times but this would incur additional gas costs and even with multiple calls a portion of the funds in the contract will forever remain locked into it.

Recommendation: Add duplicate check for the team wallets on the constructor like so:

```
@@ -29,7 +29,8 @@ contract DevDistribute {
             address wallet = _wallets[i];
             if (wallet == address(0)) revert ZeroAddress();
             shares[wallet] = share;
-            _teamWallets.add(wallet);
+            bool added = _teamWallets.add(wallet);
+            if (!added) revert Duplicate();
             totalPercents += share;
         }
         if (totalPercents != 100) revert Prohibited();
```

Resolution: Fixed in 4114583

[M-3] Unreliable rewards segregation mechanism may lead to incorrect reward allocation

Context: FluxHub.sol#L290-L299

Description: The Element 369 protocol's rewards distribution mechanism allows NFT holders who retain their NFTs throughout a full 777-day reward cycle to claim rewards from the 777 Treasury. This treasury is intended to be filled with rewards from Flux protocol's 777-day reward distributions. However, the current implementation for segregating these rewards is unreliable and may lead to incorrect reward allocation.

The mechanism operates as follows: once a 777-day reward cycle concludes, users have a 3-day window to initiate rewards claim from the Flux protocol. All rewards claimed during this 3-day window are then added to the 777 Treasury. If no claims are made within this window, any subsequent claims (e.g., on the 4th day) result in the accumulated rewards being distributed to the regular treasury instead of the 777 Treasury:

```
function _addToTreasury(uint256 amount) internal {
    uint256 amountToAdd = _getTreasuryAmount(amount);
    uint32 daysPassed = Time.daysSince(FLUX_START_DATE) + 1;
    uint32 cycles = daysPassed / 777;
    if (cycles > 0 && daysPassed <= cycles * 777 + DAY_777_OFFSET) { //@audit
     ↪  Possible misallocation of rewards
        treasury777Balance += amountToAdd;
    } else {
        treasuryBalance += amountToAdd;
    }
}
```

Given the lengthy duration of the 777-day cycle (~2 years), this approach is problematic and can lead to rewards being misallocated. The Flux protocol does not differentiate the source of the rewards being claimed. Therefore, when a user claims rewards, they receive all accumulated rewards up to that point without distinction.

The following are examples for problematic scenarios that may arise due to the current implementation:

Example scenario #1:

1. The 97th 8-day cycle in Flux ends on day 776 and no one claims the rewards on this day.
2. The first 777-day cycle completes on day 777 and a user claims the rewards.

As a result, rewards from the 8-day cycle (ending on day 776) get incorrectly added to the 777 Treasury, leading to misallocation.

Example scenario #2:

1. The first 777-day cycle ends on day 777.
2. No one claims the rewards within the 3-day window (days 778-780).
3. On day 781 (the 4th day), a user claims the rewards.

In this scenario, rewards intended for the 777-day cycle are added to the regular treasury instead of the 777 Treasury, causing an incorrect distribution of rewards.

Unfortunately, due to the lack of a mechanism within the Flux protocol to identify the source of rewards, it is not possible to segregate them accurately. Additionally, these specifics are not documented in the project whitepaper, leading to confusion and potential loss of trust in the protocol.

Recommendation: 1. Clearly outline the 777-day reward cycle distribution mechanism in the project whitepaper. This will improve transparency and set accurate expectations for users. 2. Revise reward segregation approach and consider restructuring the tokenomics to allow users to voluntarily opt-in as long-term holders by minting NFTs, thus clearly segregating 777-day cycle rewards from regular rewards.

Resolution: Acknowledged

[M-4] Precision Loss in reward calculations

Context: Element369HolderVault.sol#L403-L404

Description: The `_processTokenPool()` function is responsible for calculating the token rewards per share and updating the reward pool based on this calculated share. However, the current implementation suffers from precision loss due to integer division. Specifically, `share` is calculated as `amount / (totalMultipliers * 2)`, which truncates any remainder in the division, resulting in lower-than-expected share values:

```
function _processTokenPool(address token, uint256 amount, uint256 totalMultipliers)
    internal
    returns (uint256 share)
{
    share = amount / (totalMultipliers * 2); //@audit Precision loss
    totalTokenPool[token] += share * totalMultipliers * 2;
}
```

The issue becomes more pronounced if more tokens are minted and the `totalMultipliers` value increases, as this further divides `amount` and exacerbates precision loss, resulting in even greater inaccuracies in reward distribution.

Recommendation: To avoid precision loss, introduce a scaling factor (e.g., `1e18`) in the calculation. Multiply `amount` by this factor before dividing by `totalMultipliers * 2`, then adjust subsequent calculations to account for this scaling. This approach will preserve accuracy in the rewards calculation.

Resolution: Fixed in 13ac104

[M-5] Ineffective TWAP protection

Context: FluxHub.sol#L364

Description: The protocol employs TWAP checks for UniswapV3 swaps to protect against price manipulation. However, a single TWAP configuration is used for all pools, regardless of their volatility profiles:

```
uint32 public deviation = 2000;

// ...

function _twapCheck(...) internal view {
    // ...
    uint256 lowerBound = (twapAmountOut * (10000 - deviation)) / 10000;

    if (minAmountOut < lowerBound) revert TWAP();
}
```

This creates an inefficiency: the configuration must be set conservatively to prevent unwanted reverts with more volatile pools, but this reduces the effectiveness of the TWAP protection for less volatile pools. The one-size-fits-all approach to the TWAP protection configuration results in less optimal protection for certain pools, especially those that require tighter price control due to lower volatility.

Recommendation: Implement per-pool TWAP configurations, allowing thresholds to be customized based on the volatility of each pool.

Resolution: Fixed in 0d74346

[M-6] ERC777 hook exploit or blacklisted account could DoS token distribution

Context: DevDistribute.sol#L51

Description: The `DevDistribute` smart contract is designed to distribute tokens to team wallets for any ERC20 token. However, there are two vectors that can lead to a denial of service (DoS) in the `distributeToken()` function:

1. **ERC777/ERC20 Tokens with Hooks:** A malicious team wallet can revert the transaction in its hook during the token transfer. This would lead to a DoS attack. While there is no direct financial incentive for the attacker, the attack can halt all token payments for this particular token. The impact is limited to tokens that support hooks.
2. **Blacklisted team wallet:** If a team wallet is blacklisted for a particular ERC20 (e.g. tokens like USDC implement blacklists), the transfer to that wallet will revert, causing the entire distribution to fail for that token. As a result, all team wallets will be unable to receive their share of tokens for the affected ERC20.

The current implementation is vulnerable in both cases due to the sequential transfer logic inside a `for` loop, where any failed transfer will block the remaining transfers:

```
function distributeToken(address[] calldata tokenAddresses) external {
    if (!isTeamWallet(msg.sender)) revert Unauthorized();
    for (uint256 i = 0; i < tokenAddresses.length; i++) {
        IERC20 token = IERC20(tokenAddresses[i]);
        uint256 availableBalance = token.balanceOf(address(this));
        if (availableBalance < minAmount) revert InsufficientBalance();
        for (uint256 j = 0; j < _teamWallets.length(); j++) {
            address wallet = _teamWallets.at(j);
            uint256 share = availableBalance * shares[wallet] / 100;
            token.safeTransfer(wallet, share); //@audit DoS in case of blacklisted
             ↪  wallet or malicious ERC777 hook revert
        }
    }
}
```

A workaround for the blacklisted wallet scenario is for them to change their wallet via the `changeWallet` function. However, until they do, all payments for the token will be blocked.

Recommendation: Implement a pull over push pattern where team wallets must claim their tokens instead of tokens being transferred to them upon distribution. This would avoid potential DoS situations caused by reverts within hooks or failed transfers due to blacklists.

Resolution: Acknowledged

[M-7] NFT prices do not match the documentation

Context: Element369NFT.sol#L102-L107

Description: As per the project documentation, the mint cost for an Element 369 NFT is 369M TITANX:

$$Mint\ Cost = 369M\ TITANX$$

However, the implementation uses different prices for the tiers from the one specified in the docs:

```
tierPrices[1] = 100_000_000 ether;
tierPrices[2] = 100_000_000 ether;
tierPrices[3] = 1_000_000_000 ether;
tierPrices[4] = 1_000_000_000 ether;
tierPrices[5] = 10_000_000_000 ether;
tierPrices[6] = 10_000_000_000 ether;
```

This discrepancy may lead to confusion and may cause the protocol tokenomics to deviate from the intended one due to the different pricing.

Recommendation: Make sure the prices in the implementation match those specified in the project documentation.

Resolution: Fixed in 7952425

Low Risk

[L-1] The `originCheck` non-EOA caller protection will become ineffective with EIP-7645

Context: FluxHub.sol#L77

Description: The `FluxHub` contract currently uses an `originCheck` modifier to prevent non-EOA interactions with `distributeTreasury()` and `distribute777Treasury()`:

```
modifier originCheck() {
    if (address(msg.sender).code.length != 0 || msg.sender != tx.origin) revert
     ↪  Prohibited();
    _;
}
```

The first condition can easily be circumvented by calling the function from within a new contract constructor since during contract construction `code.length` is equal to 0.

The second condition ensures that `msg.sender` is equal `tx.origin` which is only true for EOAs. However, if EIP-7645 is accepted, `msg.sender` and `tx.origin` would become aliases, rendering this check ineffective.

Recommendation: It is advisable to consider alternative methods to restrict contract-based interactions. For example, if the intended use of this modifier is flashloan protection, consider using slippage protection mechanisms and/or TWAP-based checks.

Resolution: Acknowledged

Informational

[I-1] Superfluous empty string check for `baseURI` in `Element369NFT.tokenURI()`

Context: Element369NFT.sol#L361

Description: The implementation of `Element369NFT.tokenURI()` checks if `baseURI` is empty and if so returns an empty string:

```
return bytes(baseURI).length != 0 ? string(abi.encodePacked(baseURI,
 ↪  getNftTier(tokenId).toString(), ".json")) : "";
```

However, `baseURI` is set only via the contract constructor or `setBaseURI` and in both cases the configured value is checked to be a non-empty string. Thus, the empty string check is superfluous and should be removed.

Recommendation: Remove the `baseURI` empty string check in `tokenURI`:

```
@@ -357,8 +357,7 @@ contract Element369NFT is ERC721A, Ownable, IERC165 {

 function tokenURI(uint256 tokenId) public view virtual override returns (string
  ↪  memory) {
     if (!_exists(tokenId)) revert URIQueryForNonexistentToken();
-    return bytes(baseURI).length != 0 ? string(
-        abi.encodePacked(baseURI, getNftTier(tokenId).toString(), ".json")
-     ) : "";
+    return string(
+       abi.encodePacked(baseURI, getNftTier(tokenId).toString(), ".json")
+     );
 }

 function supportsInterface(bytes4 interfaceId) public view virtual override(IERC165,
  ↪  ERC721A) returns (bool) {
```

Resolution: Fixed in 7ca8311

[I-2] Off-by-one error in `Element369NFT.getTotalNftsPerTiers()`

Context: Element369NFT.sol#L227

Description: The natspec for the `getTotalNftsPerTiers()` function in `Element369NFT` says:

```
/// @notice Returns the total number of NFTs per tier.
/// @return total An array where each index corresponds to the total NFTs for a
 ↪  specific tier.
```

However, the implementation subtracts 1 by the NFT tier when aggregating the NFT tier count:

```
total[_getNftTier(_packedTokenData[tokenId]) - 1]++;
```

This results in an off-by-one error where 0th index would be Tier 1, 1st index would be Tier 2, etc.

This is also inconsistent with the behavior of the `getNftTier()` function which does not modify the returned NFT tier.

Recommendation: Do not subtract 1 from the NFT tier to align the array indexes with the tiers.

Alternatively, update the natspec to document this behavior and align the implementation of `getNftTier()` with `getTotalNftsPerTiers()`.

Resolution: Acknowledged

[I-3] Missing or non-informative event emissions

Context: Multiple Contracts

Description: Throughout the codebase, important state-changing functions do not emit events. Here are examples of such functions:

- `Element369NFT.setProtocolAddresses()`
- `Element369NFT.setSecondsAgo()`
- `Element369NFT.setDeviation()`
- `Element369NFT._setNftData()`
- `Element369NFT._updateBurnCycle()`
- `DevDistribute.distributeToken()`
- `DevDistribute.changeWallet()`
- `FluxHub.startStake()`

Additionally, some functions emit events but without any meaningful arguments, resulting in non-informative events that do not provide useful tracking information:

- `AuctionEntered()`
- `AuctionsClaimed()`
- `RewardsClaimed()`

Recommendation: Consider emitting informative events from important state-updating functions to more easily enable off-chain monitoring for the protocol.

Resolution: Partially fixed in a7f473c

[I-4] Use constants instead of literals

Context: Element369NFT.sol#L117, Element369NFT.sol#L131, Element369NFT.sol#L204, Element369NFT.sol#L463, FluxHub.sol#L249, FluxHub.sol#L284, FluxHub.sol#L364, DevDistribute.sol#L35, DevDistribute.sol#L50, Element369HolderVault.sol#L414, Element369HolderVault.sol#L415

Description: The literal following literals are used throughout the codebase:

- 10000 - when calculating basis points (bps)

- `100` - when calculating percentages

It is a good practice to use constants rather than literals to make the code easier to understand and maintain.

Recommendation: Define the following constants and use them in all applicable contexts:

- `uint32 constant BPS = 10_000`
- `uint32 constant PERCENTAGE = 10_000`

Resolution: Fixed in 24bec06

[I-5] Inconsistencies with the whitepaper

Context: N/A

Description: 1. The protocol employs an incentive fee of 0.3% (configurable by the protocol owner) to users who call the `FluxHub.claimRewards()` function. But the whitepaper of the protocol contains no mention of this type of fee. 2. As per the whitepaper, NFT rewards are distributed every 11 days, with no mention of a cycle-based approach for NFT backing claims. However, in the implementation, the reward and backing claims share the same mechanics, meaning NFT backing claims are also bound to the 11-day reward cycles. In practice, this means users can only claim NFT backing up to the end of the last completed reward cycle at the time they burn their NFT.

Recommendation: Review and update contracts to accurately reflect the project's intended design as per the whitepaper, or vice versa, to maintain consistency and clear expectations.

Resolution: Acknowledged