marchev

Flux
Security Review

Version 1.2

Reviewed by:

Martin Marchev

September 14, 2024

Table of Contents

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 Flux

Flux is an ERC-20 token with deflationary tokenomics, featuring a minting auction, buy-and-burn mechanism and a staking contract that provides rewards to stakers.

2.2 Synopsis

Project Name	Flux
Project Type	ERC-20
Repository	flux-contracts
Commit Hash	e6acb219...abfa8f1c
Review Period	28 August 2024 - 11 September 2024

2.3 Issues Found

Severity	Count
Critical Risk	0
High Risk	6
Medium Risk	8
Low Risk	4
Informational	10
Total Issues	**28**

3 Findings

High Risk

[H-1] Malicious actors can exploit `FluxAuction` distribution incentivization through self-sandwich attacks

Context: FluxAuction.sol#L285, SwapActions.sol#L264

Description: The `FluxAuction` contract includes a mechanism to incentivize users to distribute the accumulated TitanX by calling the `FluxAuction.distribute()` function. Users are rewarded with 1.5% of the distributed TitanX amount. However, the implementation allows malicious actors to exploit this distribution mechanism by performing a self-sandwich attack, resulting in significantly higher profits than the intended 1.5% incentivization.

The vulnerability arises because the `FluxAuction.distribute()` function allows the caller to specify the slippage tolerance for the TitanX/Blaze swap. By setting a 100% slippage tolerance, a malicious actor can manipulate the swap in their favor and profit more than the intended reward at the expense of the amount of Inferno that will be burned, essentially impacting protocol tokenomics.

Example:

Prerequisite: The TitanX/Blaze pool has the following liquidity:

- 79,000,000,000 TitanX
- 32,300 Blaze

1. Users deposit $10,000 worth of TitanX into `FluxAuction`. Thus, $5,000 worth of TitanX will be distributed and $500 worth will be swapped for Inferno and burned.
2. Malicious actor sees the accumulated TitanX and performs the following self-sandwich attack:
    - Swap 32,986,376,889 TitanX (~$15,600) for Blaze in the Uniswap V2 pool, receiving 9,494 Blaze.
    - Call `FluxAuction.distribute()` with no slippage protection.
    - Swap back the 9,494 Blaze for 33,375,634,704 TitanX (~$15,804).

As a result, the malicious actor makes a profit of $204, significantly more than the $75 (1.5% of $5,000) they would have earned through the regular incentivization mechanism.

Recommendation: To mitigate this vulnerability, the protocol should use a TWAP mechanism to prevent manipulation during the TitanX/Blaze swaps. This will ensure that the distribution process remains fair and that the incentivization mechanism cannot be exploited for undue profit.

Resolution: Fixed in 67be8cc

[H-2] Incorrect Flux-to-share ratio calculation results in users receiving fewer shares

Context: Staking.sol#L213-L218

Description: The `Staking.getFluxToShareRatio()` function calculates the Flux-to-share ratio incorrectly. This leads to users receiving fewer shares than intended according to the protocol's

tokenomics.

As per the documentation, the intended behavior is that in the first 8-day period, 1 Flux should equal 1 Share. After each 8-day period, the value of 1 Share in Flux should decrease by 1.26%. However, the current implementation erroneously calculates the ratio as if 1 Flux would receive 1.26% more shares, which is mathematically incorrect due to an improper basis for the calculation.

The correct formula should be based on the fact that:

$$1\ Share = (1 - 0.0126)\ Flux = 0.9874\ Flux$$

Thus:

$$1\ Flux = \frac{1}{0.9874}\ Shares = 1.0127607859\ Shares$$

This results in a user receiving 1.27607859% more shares, not exactly 1.26% more shares.

The correct Flux-to-share calculation should follow an exponential growth formula for the share distribution over time, simplified as:

$$1\ Flux = (\frac{100}{100 - 1.26})^{P}\ Shares$$

where P represents the number of 8-day periods that have passed.

This incorrect calculation leads to users receiving fewer shares than intended, which breaks the tokenomics model of the protocol and creates a disadvantage for users staking their Flux tokens. Given the exponential nature of the growth, over time this discrepancy would accumulate, leading to substantial under-distribution of shares.

Recommendation: The calculation of the Flux-to-share ratio should be corrected by implementing the proper exponential growth formula:

$$1\ Flux = (\frac{100}{100 - 1.26})^{P}\ Shares$$

Resolution: Fixed in 5d91b75

[H-3] Incorrect `totalShares` update during unstaking leads to reward dilution and irrecoverable funds

Context: Staking.sol#L140

Description: There is an issue in the `FluxStaking` contract where `totalShares` is incorrectly updated during the unstaking process.

The `totalShares` state variable is decreased by the value of the `shares` variable. However, the `shares` variable in this context represents the number of shares the user has in the 777 Voluntary pool:

```
(uint160 shares,) = voluntary.record(_tokenId);
```

However, if the user has not participated in this pool, the shares value will be `0`. As a result, when the contract subtracts `shares` from `totalShares` during unstaking, the `totalShares` value is not updated properly.

Since `totalShares` remains incorrectly inflated when users with no shares in the 777 Max Voluntary staking pool unstake, all future reward distributions will be diluted. This leads to inaccurate reward calculations for all stakers, reducing the rewards they should rightfully receive. Moreover, the rewards proportional to the user's actual share remain irrecoverably stuck in the contract.

Recommendation: Update the `totalShares` value by using the correct `record.shares` value.

Resolution: Fixed in 9a6a798

[H-4] Staking rewards are only distributed if there is interaction with the protocol on reward days

Context: Staking.sol#L162-L175

Description: There is an issue in the `FluxStaking` contract where staking rewards are not distributed unless there is an interaction with the protocol (such as deposits or claims) on the specific days when rewards should be distributed. While rewards are not lost, they cannot be claimed until the next reward cutoff point, forcing users to wait unnecessarily long periods to access their staking rewards.

Example scenario:

1. Alice stakes for the maximum staking period of 2888 days, thus becoming part of the 777 Club.
2. On day 777, there are no interactions with the protocol - no deposits and no claims.
3. On day 778, Alice attempts to claim her rewards but is unable to do so because the reward distribution is dependent on interaction with the protocol on day 777. As a result, Alice must now wait until day 1554 to claim her 777-day rewards.

The underlying issue is that while the protocol might be highly active in the early stages with regular interactions ensuring timely reward distribution, this problem will become more pronounced as time progresses, especially for longer staking periods.

If the protocol becomes less active, users may be unable to claim their rewards unless they interact with the protocol on the exact day rewards are distributed.

Recommendation: To resolve this issue, the reward distribution should be decoupled from user-triggered interactions and be based on a mathematical model that continuously calculates rewards, similar to how distribution is handled in the SushiSwap MasterChef contract. In SushiSwap, rewards are calculated over time using block numbers, ensuring users can claim rewards at any time without needing interaction on specific days. Implementing a similar approach will ensure rewards are consistently distributed over time, allowing users to claim their rewards at any point after the distribution date, regardless of protocol activity.

Resolution: Fixed in a8c355f

[H-5] Incorrect distribution of staking rewards leading to imbalanced pool allocations

Context: Staking.sol#L186-L196

Description: The current implementation of staking reward distribution in the `FluxStaking` contract does not follow the intended tokenomics, leading to incorrect and imbalanced distribution across reward pools.

According to the intended design, 40% of each deposit made via the Flux auction should be allocated to the staking contract as rewards, and these rewards should be split immediately across four staking pools in the following proportions:

- 38% to the 8-day pool
- 30% to the 28-day pool
- 22% to the 88-day pool
- 10% to the 777-day pool

The distribution should occur with every deposit, meaning that rewards are continuously and proportionally distributed to all pools. However, the current implementation incorrectly accumulates deposits in the contract and only distributes them on specific reward days, causing severe misallocation.

Example:

1. A total of 1,000,000 TitanX is deposited into the system. For simplicity, let's assume no other deposits are made.
2. On day 8, 38% of the entire contract balance (380,000 TitanX) is distributed to the 8-day pool.
3. From day 9 to 15, no rewards are distributed to any pools.
4. On day 16, another 38% of the remaining contract balance (approximately 235,600 TitanX) is distributed to the 8-day pool.
5. No rewards are distributed to the 28-day, 88-day, or 777-day pools during this time.

As a result, the 8-day pool receives the majority of the accumulated TitanX, while the other pools remain underfunded, leading to severe imbalance in reward allocation. This incorrect behavior deviates from the intended tokenomics, where all pools should receive a proportional distribution of rewards with every deposit, regardless of the day.

Over time, this flaw undermines the fairness of the staking mechanism, as users staking in longer-term pools are disadvantaged compared to those in shorter-term pools.

Recommendation: The distribution logic should be corrected to ensure that rewards are distributed to all pools immediately after every deposit, according to the intended proportions:

- 38% to the 8-day pool
- 30% to the 28-day pool
- 22% to the 88-day pool
- 10% to the 777-day pool

This change will ensure that rewards are fairly distributed across all pools and align with the protocol's tokenomics.

Resolution: Fixed in a8c355f

[H-6] Incorrect implementation of buy-and-burn mechanism deviates from intended tokenomics

Context: FluxBuyAndBurn.sol#L119-L123

Description: The current implementation of the Flux buy-and-burn mechanism deviates from the intended and documented behavior, breaking a core invariant of the protocol. According to the documentation, the intended process should follow these steps:

1. TitanX should be swapped for Inferno.
2. 20% of the Inferno amount should be burned directly.
3. The remaining 80% of the Inferno amount should be swapped for Flux, which is then burned.

However, the current implementation swaps TitanX for Inferno and then swaps the entire Inferno amount for Flux, burning the Flux tokens:

```
function swapTitanXForFluxAndBurn(uint32 _deadline) external intervalUpdate {
    Interval storage currInterval = intervals[lastIntervalNumber];
    if (currInterval.amountBurned != 0) revert IntervalAlreadyBurned();

    currInterval.amountBurned = currInterval.amountAllocated;

    uint256 incentive = wmul(currInterval.amountAllocated, INCENTIVE_FEE);

    uint256 titanXToSwapAndBurn = currInterval.amountAllocated - incentive;

    uint256 infernoAmount = _swapTitanXForInferno(titanXToSwapAndBurn, _deadline);

    uint256 fluxAmount = _swapInfernoForFlux(infernoAmount, _deadline);

    burnFlux();

    titanX.safeTransfer(msg.sender, incentive);

    emit BuyAndBurn(titanXToSwapAndBurn, fluxAmount, msg.sender);
}
```

This discrepancy undermines the protocol's intended tokenomics by not adhering to the designed split between burning Inferno directly and converting it to Flux before burning.

Recommendation: The implementation should be updated to match the documented behavior. Specifically:

1. Swap TitanX for Inferno as currently implemented.
2. Burn 20% of the Inferno amount immediately.
3. Swap the remaining 80% of Inferno for Flux and burn the Flux tokens.

This will ensure the protocol adheres to its intended tokenomics, maintaining the designed deflationary pressure on both tokens.

Resolution: Fixed in dcd2776

Medium Risk

[M-1] Incorrect reward distribution allows users to claim rewards for previous periods they did not stake in

Context: Staking.sol#L167-L198

Description: There is an issue in the `FluxStaking` contract where users can stake in a new period and still receive rewards for previous periods they did not participate in.

This happens when a reward distribution covers a previous period, the next reward distribution occurs in the current period and a user stakes between the two distributions. This allows users to unfairly claim rewards for which they are not eligible.

Example Scenario:

1. On Day 1, Alice stakes 50 FLUX.
2. On Day 7, 1000 TITANX is distributed as rewards to the staking contract.
3. On Day 9, Bob stakes 50 FLUX during the second 8-day period.
4. On Day 10, another 1000 TITANX is distributed for the second period.
5. On Day 16, both Bob and Alice claim their rewards.

The expected behavior is that Bob should only be eligible to receive rewards from the Day 10 distribution (500 TITANX), and Alice should receive both the Day 7 and Day 10 rewards (1500 TITANX in total).

The actual behavior is that Bob incorrectly receives 1000 TITANX, despite only staking on Day 9 (after the first 8-day reward period is over). Alice, who should have received 1500 TITANX, only receives 1000.

This is the distribution logic in the `updateRewardsIfNecessary()` function:

```
bool distributeDay8 = (currentDay / 8 > lastDistributedDay / 8);
bool distributeDay28 = (currentDay / 28 > lastDistributedDay / 28);
bool distributeDay88 = (currentDay / 88 > lastDistributedDay / 88);
bool distributeDay777 = (currentDay / 777 > lastDistributedDay / 777);

// Distribute for the 8-day pool if necessary
if (distributeDay8) _updateRewards(POOLS.DAY8, toDistribute);

// Distribute for the 28-day pool if necessary
if (distributeDay28) _updateRewards(POOLS.DAY28, toDistribute);

// Distribute for the 88-day pool if necessary
if (distributeDay88) _updateRewards(POOLS.DAY88, toDistribute);
```

Rewards are distributed for the corresponding reward period once the required number of days have elapsed. The problem is that this function is called when claiming and distributing rewards but not when staking. In the example above, this will cause the `totalShares` number to increase when Bob stakes without distributing the rewards which will cause his shares to be incorrectly in the calculation for the previous reward period during the next rewards distribution.

This flaw dilutes the rewards for earlier stakers and allows users to unfairly benefit from rewards that they should not be entitled to. While the issue only occurs when reward distributions span multiple periods and new users stake before the second distribution, it still leads to an imbalance in the reward system, impacting the fairness of the staking mechanism.

Recommendation: Update the reward calculation logic to ensure that users who stake in the current period are only eligible for rewards from that period onwards. This could be achieved by calling `updateRewardsIfNecessary()` at the beginning of the `FluxAuction.stake()` function.

Resolution: Fixed in e4c6fb7

[M-2] Incorrect handling of interval updates in `FluxBuyAndBurn._intervalUpdate()` breaks the buy-and-burn mechanism

Context: FluxBuyAndBurn.sol#L281

Description: There is an issue with the handling of interval updates in the `FluxAuction._intervalUpdate()` function that disrupts the protocol's tokenomics.

When TitanX is distributed to the contract via `distributeTitanXForBurning()` , the implementation mistakenly records that a burn event has happened, even though only an intervals allocation update has occurred. This leads to a scenario where the unburned TitanX from previous intervals is incorrectly marked as burned, causing only the TitanX from the intervals after the last distribution to be burned during the next `swapTitanXForFluxAndBurn()` call.

Example:

1. The buy-and-burn mechanism starts at 5:00 PM.
2. At 5:05 PM (interval #1), Alice deposits and distributes 1,400,000,000 TitanX.
3. The next day, at 4:40 PM (interval = 95), Alice deposits another 1,400,000,000 TitanX and distributes them. 24 hours have not passed since the last snapshot is taken, thus this amount is not accounted for the current day allocation. However, intervals update operation is performed.
4. At 4:50 PM (interval = 96), the `swapTitanXForFluxAndBurn()` function is called. Only the TitanX amount for the last interval is burned (2,068,500), instead of the entire day's allocation (198,576,000).

This issue occurs because during the `distribute()` function call, the following line incorrectly marks the intervals update as a burn:

```
lastBurnedIntervalStartTimestamp = _lastIntervalStartTimestamp +
↪  (uint32(_missedIntervals) * INTERVAL_TIME);
```

This causes the system to incorrectly register that a burn event has occurred even though it hasn't,

leading to inaccurate burning and disruption of the protocol's tokenomics.

Recommendation: do not update `lastBurnedIntervalStartTimestamp` when `_intervalUpdate()` is triggered via TitanX distribution operation.

Resolution: Fixed in f9ed8fa

[M-3] Emission rate incorrectly updated after periods of inactivity, potentially breaking tokenomics

Context: FluxAuction.sol#L227-L232

Description: The `FluxAuction._updateAuction()` function is designed to update the token emission rate weekly, reducing it by a fixed percentage (1.2%) each week. The current implementation checks if a new week is detected by comparing `a.lastWeekUpdate` to `currWeek`. If they differ, the emission is reduced by 1.2% based on the last emission rate and `a.lastWeekUpdate` is updated to the current week:

```
if (a.lastWeekUpdate != currWeek) {
    percentagesToDrop = WEEKLY_EMISSION_DROP;
    a.lastWeekUpdate = currWeek;
}

a.fluxToEmit = uint160(sub(a.fluxToEmit, int256(wmul(a.fluxToEmit,
↪  percentagesToDrop))));
```

While this approach works correctly when `a.lastWeekUpdate` is the previous week, it fails when there are one or more weeks of inactivity between `a.lastWeekUpdate` and `currWeek`. In such cases, the emission rate is only reduced by 1.2% once, rather than being compounded for each week of inactivity. This results in an incorrect emission rate, which could lead to significantly higher token emissions than intended.

Given that the protocol is designed to run for 8 years (a long duration in the DeFi space), periods of inactivity are likely to occur at some point. When this happens, the incorrect emission rate calculation could break the intended tokenomics.

Recommendation: The most effective solution is to replace the current emission reduction logic with an exponential decay function that accurately calculates the emission rate for the current week, regardless of activity:

$$C = I \times (1 - \frac{1.2}{100})^W$$

$$C - Current\ Week\ Emission\ Rate$$

$$I - Initial\ Emission\ Rate$$

$$W - Current\ Week\ Number\ (0 - based)$$

This ensures that the emission rate is correctly reduced for each week, maintaining the intended tokenomics.

Resolution #1: Fixed in ae383d4

Post-Fix Review #1: The formula is incorrect. The applied formula in the fix is:

$$I \times (1 - \frac{1.2}{1000}^{W})$$

However, the correct formula is:

$$I \times (1 - \frac{1.2}{100})^{W}$$

It should be implemented like this:

```
uint160 emitted = uint160(wmul(AUCTION_EMIT, wpow(sub(1e18, WEEKLY_EMISSION_DROP),
↪  currWeek, WAD)));
```

Resolution #2: Fixed in a514d0f

Post-Fix Review #2: Fix confirmed

[M-4] `FluxAuction.addLiquidityToInfernoBlazePool()` susceptible to griefing and multi-block MEV

Context: SwapActions.sol#L131, SwapActions.sol#L135, OracleLibrary.sol#L77

Description: The `FluxAuction.addLiquidityToInfernoBlazePool()` function is vulnerable to both griefing and multi-block MEV attacks due to the Inferno/TitanX pool's low observation cardinality of 1. This cardinality results in the TWAP being based on only one historical price, making it possible to disrupt the liquidity bootstrapping process or manipulate the TWAP.

In `getFluxQuoteForInferno()` the `OracleLibrary.getOldestObservationSecondsAgo()` function is used to retrieve the age of the oldest observation. UniswapV3 observations are updated when a state-changing operation (e.g. swaps or adding/removing liquidity) occurs in the pool. If a malicious actor frontruns the `addLiquidityToInfernoBlazePool()` function call by performing a swap, they trigger an observation write within the same block. With an observation cardinality of 1, this causes the oldest observation to be 0 seconds old since it is written in the same block. `FluxAuction.getFluxQuoteForInferno()` uses `OracleLibrary.consult()` to calculate the actual TWAP but this function has the following precondition:

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

Thus, by frontrunning the `addLiquidityToInfernoBlazePool()` function call and executing a swap, a malicious actor would cause the function to revert.

Example scenario:

1. Alice calls `addLiquidityToInfernoBlazePool()` to bootstrap liquidity.
2. Bob frontruns Alice's transaction with a swap in the TitanX/Inferno UniV3 causing an observation write in the same block.

3. Alice's transaction executes after Bob's but reverts because `OracleLibrary.getOldestObservationSecondsAgo()` returns 0, causing `OracleLibrary.consult()` to fail.

Additionally, due to the low observation cardinality, the function is also susceptible to multi-block MEV where a malicious actor could manipulate the price in the pool in block `N` and then ordering the vicim transaction at the top of the block in `N+1` and backrunning it to avoid arbitrage losses. This would effectively allow a malicious MEV searcher to manipulate the TWAP which is used to calculate the slippage protection for the TitanX/Inferno swap and effectively perform a multi-block sandwich attack. The likelihood is low though.

Recommendation: Increase the observation cardinality for the Inferno/TitanX pool to improve TWAP accuracy and reduce susceptibility to griefing attacks and manipulation.

Resolution: Fixed in 3b58b77

[M-5] Auction cutoff time is not precisely 2 PM UTC due to relative timestamp usage in the deployment script

Context: DeployFlux.s.sol#L16

Description: The `Flux` deployment script uses a relative timestamp to set the `_auctionStartTimestamp` parameter, specifically `block.timestamp + 2 hours`. This approach cannot guarantee that the auction will start precisely at 2 PM UTC as it assumes the transaction to deploy the contract will be executed exactly at 12:00:00 PM UTC. However, in reality, the `block.timestamp` can vary due to factors like network congestion or delays, leading to an imprecise auction start time.

This imprecision will cause various time-based calculations in the protocol to be incorrect as some of these rely on the auction start time being precisely at 2 PM UTC. For example, `Time.dayCountByT()` implies an exact 2 PM UTC start time while `Time.weekSince()` simply assumes that the auction cutoff time is 2 PM UTC. This will lead to incorrect results and time-based inconsistencies within the protocol.

Recommendation: Use an absolute Unix timestamp for the auction start time rather than a relative one.

Additionally, it is advisable to add validation within the `FluxAuction` constructor to ensure that the auction start time is exactly 2 PM UTC, e.g.:

```
@@ -99,6 +99,9 @@ contract FluxAuction is SwapActions {
         FluxBuyAndBurn _bnb,
         address _titanXInfernoPool
     ) SwapActions(_flux, _titanX, _inferno, _blaze, _blazeTitanXPool,
      ↪  _titanXInfernoPool, _owner) {
+        if ((_startTimestamp - 14 hours) % 1 days != 0) {
+            revert FluxAuction__AuctionsMustStartAt2PMUTC()
+        }
         auction.startTimestamp = _startTimestamp;
         auction.fluxToEmit = AUCTION_EMIT;
```

Such validation will prevent any deviation from the required start time, ensuring that all time-based calculations within the protocol remain accurate and consistent.

Resolution: Fixed in ff9e3cb

[M-6] Potential manipulation of Flux/Inferno pool due to delayed liquidity provision

Context: FluxAuction.sol#L161-L198

Description: A malicious actor could bootstrap the Flux/Inferno Uniswap V3 pool with a heavily skewed price if the protocol does not accumulate the required 50B TitanX tokens on the first auction day. The bootstrapping of the Flux/Inferno pool is a two-step process:

1. The Flux/Inferno pool is created during the `Flux` token deployment with an initial price;
2. The actual liquidity provision is triggered later once the protocol accumulates at least 50B TitanX;

While the likelihood is low, if the protocol fails to accumulate the necessary TitanX on day one, Flux tokens will begin circulating on day two without adequate liquidity in the Flux/Inferno pool. A malicious actor could then exploit this situation by adding very shallow liquidity to the pool at the initial price. They could manipulate the price in a direction that benefits them and after doing so add deep liquidity at the manipulated price to maximize their profits.

Example scenario:

1. Alice realizes the protocol hasn't accumulated the required 50B TitanX on the first auction day, meaning the Flux/Inferno pool will have no liquidity when Flux tokens start circulating on day two.
2. On day two, Alice adds a small amount of liquidity to the pool at the initial price.
3. With the pool's shallow liquidity, Alice manipulates the price by executing trades, skewing it in her favor.
4. After manipulating the price, Alice adds deep liquidity at the skewed rate, positioning herself to profit as other users trade against this manipulated price.

This flaw could be exploited in various ways. E.g.:

- The malicious actor could force the protocol to add the accumulated liquidity at an unfavorable, manipulated price, locking in a distorted market price;
- The attacker could also grief the liquidity bootstrapping process by manipulating the price beyond the slippage tolerance, preventing the protocol from adding liquidity altogether;

Recommendation: Delaying the creation of the Uniswap V3 pool is not recommended as this could allow a malicious actor to frontrun the pool creation and create it with an unfavorable initial price. The best course of action is to ensure that the liquidity is bootstrapped on the first auction day. If the protocol does not accumulate the required TitanX by the end of the day, the protocol team should contribute its own funds to the auction to ensure the pool is bootstrapped before Flux tokens are in circulation. This proactive measure will protect against potential manipulation and maintain the integrity of the pool's pricing.

Resolution: Acknowledged

[M-7] Immediate burning of Inferno by `FluxAuction.distribute()` deviates from intended protocol design

Context: FluxAuction.sol#L282-L283

Description: The `FluxAuction.distribute()` function immediately swaps TitanX for Inferno and burns it, rather than gradually distributing it for burning as outlined in the Flux project documentation. This immediate burn impacts the tokenomics by creating large, sudden reductions in Inferno supply, which can destabilize prices.

This behavior deviates from the intended design, which promotes gradual burning to maintain stability in the token supply and tokenomics.

Recommendation: Follow the documented approach by distributing TitanX for gradual burning rather than burning it all at once.

Resolution: Fixed in cc54baa

[M-8] TWAP-based slippage protection during deposits can lead to insufficient protection and unnecessary reverts

Context: FluxAuction.sol#L326-L327, SwapActions.sol#L179-L181, SwapActions.sol#L237-L239

Description: From the second day onwards, the `Flux.deposit()` function uses 50% of the deposited TitanX to buy Flux and stake it forever via two swaps: TitanX/Inferno and Inferno/Flux. Both swaps feature TWAP-based slippage protection, which relies on the Uniswap V3 TWAP for calculating the slippage tolerance. However, TWAP is a lagging indicator and does not react immediately to sharp price movements, leading to two potential issues during periods of high volatility:

1. **Insufficient slippage protection**: If there is a sharp price increase, the TWAP-based slippage tolerance will allow for higher-than-expected slippage, exposing the user to sandwich attacks.
2. **Unnecessary reverts**: During sharp price drops, the TWAP-based slippage protection could exceed market prices, causing deposits to revert unnecessarily and temporarily DoS user deposits.

Unlike the buy-and-burn function, which is publicly callable and may pose a conflict of interest, users have no incentive to manipulate slippage in `Flux.deposit()` and would naturally aim for the best value.

Recommendation: Replace TWAP-based slippage protection in the swaps executed in `Flux._autoBuyAndMaxStake()` with a `minAmountOut`-based slippage mechanism to ensure accurate protection during times of high volatility and prevent unnecessary reverts.

Low Risk

[L-1] Total auctions length is 2888 days instead of 2922

Context: FluxAuction.sol#L116, Staking.sol#L44

Description: The protocol documentation specifies that the total duration of the auctions should span 2922 days. However, upon reviewing the contract, it has been found that the actual implemen-

tation sets the total auctions length to 2888 days. This discrepancy of 34 days could lead to confusion among users and misalignment with expected protocol behavior, although it does not pose a significant security risk.

Recommendation: Adjust the contract logic to ensure that the total auction length aligns with the documented duration of 2922 days, thereby maintaining consistency between the implementation and the documentation.

Resolution: Fixed in 0918cb6

[L-2] Discrepancy in TitanX distribution percentages between implementation and documentation

Context: Constants.sol#L25-L30

Description: The TitanX distribution percentages implemented in the code differ from those documented in the Flux protocol's official documentation. The implemented values are:

```
uint64 constant GENESIS = 0.08e18; // 8%
uint64 constant TITAN_X_HELIOS = 0.01e18; // 1%
uint64 constant TITAN_X_DRAGON_X = 0.05e18; // 5%
uint64 constant FLUX_BUY_AND_BURN = 0.36e18; // 36%
uint64 constant BLAZE_BURN = 0.1e18; // 10%
uint64 constant REWARD_POOLS = 0.4e18; // 40%
```

However, the documentation specifies:

> ***TitanX Token Distribution from auctions :*** *10% TitanX Buys Blaze swaps to Inferno then burns it. 36% FLUX Buy & Burn 40% reward pools 4% TitanX sent to DragonX Vault 2% TitanX sent to the Helios Treasury 8% Genesis (no expectations)*

This mismatch would result in incorrect token allocations, potentially impacting the protocol's operations and economic model.

Recommendation: Update the implementation to match the documented percentages, ensuring that the Helios Treasury receives 2% and the DragonX Vault receives 4%, as intended in the documentation:

```
 uint64 constant WEEKLY_EMISSION_DROP = 0.012e18; // 1.2%
 uint64 constant WEEKLY_SHARE_DECREASE = 0.0126e18;
 uint64 constant GENESIS = 0.08e18; // 8%
-uint64 constant TITAN_X_HELIOS = 0.01e18; // 1%
-uint64 constant TITAN_X_DRAGON_X = 0.05e18; // 5%
+uint64 constant TITAN_X_HELIOS = 0.02e18; // 2%
+uint64 constant TITAN_X_DRAGON_X = 0.04e18; // 4%
 uint64 constant FLUX_BUY_AND_BURN = 0.36e18; // 36%
 uint64 constant BLAZE_BURN = 0.1e18; // 10%
 uint64 constant REWARD_POOLS = 0.4e18; // 40%
```

Resolution: Fixed in 2e6f23d

[L-3] Multiple tests are failing

Context: N/A

Description: Multiple test cases are failing in the provided repo. Tests are an essential part of the development and security cycle. Consider fixing test cases and ensuring that all tests pass and that the coverage is high.

Recommendation: Fix failing test cases and ensure that all tests pass and provide high code coverage.

Resolution: Fixed in 33b7678

[L-4] Incorrect slippage configuration used for TitanX/Inferno swaps

Context: SwapActions.sol#L202

Description: The `SwapActions._swapTitanXForInferno()` function is used to execute swaps between TitanX and Inferno tokens. However, incorrect slippage tolerance configuration is used for these swaps - `infernoToFluxSlippage` instead of the correct `titanXToInfernoSlippage`. This misconfiguration could result potentially lead to higher-than-expected slippage during TitanX/Inferno swaps.

Recommendation: Update the `SwapActions._swapTitanXForInferno()` function to use the correct `titanXToInfernoSlippage` configuration.

Resolution: Fixed in 36ea15c

Informational

[I-1] INFERNO/FLUX UniV3 pool could be created with a heavily manipulated and skewed price via a sandwich attack

Context: Flux.sol#L116

Description: The protocol calculates the initial price for the INFERNO/FLUX UniV3 pool by using the output of the `Quoter.quoteExactInputSingle()` function in `Flux._createUniswapV3Pool()` to calculate the Inferno amount for the initial price. However, the small liquidity in the TITANX/INFERNO pool (~$87K) makes it susceptible to price manipulation. A malicious actor can perform a sandwich attack by devaluing TITANX relative to INFERNO just before the `quoteExactInputSingle()` call. This manipulation results in the protocol receiving fewer INFERNO tokens than expected, leading to the INFERNO/FLUX pool being initialized with a skewed and unintended price. The attack is highly feasible due to the low liquidity and the reliance on on-chain price calculations via `quoter.quoteExactInputSingle()`.

Recommendation: To mitigate this vulnerability, the protocol should calculate the initial INFERNO/FLUX price off-chain using a reliable and secure price feed. The calculated price should then be directly passed during the pool creation, setting the sqrtX96 price manually, thereby avoiding reliance on the potentially manipulated on-chain price oracle. This approach would prevent the initialization of the pool with a manipulated price and protect against sandwich attacks.

Resolution: Acknowledged. The pool would be created via a private RPC.

[I-2] Make dependencies immutable

Context: Flux.sol#L36, Flux.sol#L38

Description: The `auction` and `staking` contract dependencies in the `Flux` contract are never changed and should thus be marked as `immutable`.

Recommendation: Apply the following fix:

```
 contract Flux is ERC20Burnable {
     //========IMMUTABLES========//

-    FluxAuction public auction;
+    FluxAuction immutable public auction;
     FluxBuyAndBurn public immutable buyAndBurn;
-    FluxStaking public staking;
+    FluxStaking immutable public staking;
     address public immutable pool;

     //===========ERRORS===========//
```

Resolution: Fixed in 0b90931

[I-3] `currentDay` in `FluxAuction._updateAction()` is set to the nth day since 2 PM on the Unix Epoch

Context: FluxAuction.sol#L219, Time.sol#L13-L18

Description: The `currentDay` variable in `FluxAuction._updateAction()` is calculated based on the number of days that have passed since 2 PM on the Unix epoch (14:00:00, Jan 1st, 1970). This occurs due to the implementation of `Time.dayCountByT()` which counts completed days starting from the Unix epoch:

```
function dayCountByT(uint32 t) internal pure returns (uint32 dayCount) {
    assembly {
        let adjustedTime := sub(t, 50400)
        dayCount := div(adjustedTime, 86400)
    }
}
```

As a result, the `currentDay` value reflects the nth day since the Unix epoch instead of the nth day since the auctions `startTimestamp`. For instance, if the auctions begin on Fri Aug 30, 2024, at 14:00:00 UTC, the `currentDay` for the first auction day will be 19965 instead of 1 (or 0, depending on whether zero-based counting is used). While this issue does not impact the functionality, it may complicate debugging, as `currentDay` is used as a key in `dailyStats`.

Recommendation: To improve clarity and align `currentDay` with the auctions timeline, consider calculating `currentDay` relative to `startTimestamp`. This adjustment will ensure that `currentDay` starts from 1 (or 0) on the first day of the auction, making debugging and interpretation of `dailyStats` more intuitive.

Resolution: Fixed in 16d2083

[I-4] Add a deadline check to `FluxAuction.deposit()` for improved gas efficiency

Context: FluxAuction.sol#L112

Description: The `FluxAuction.deposit()` function currently passes the `_deadline` argument to Uniswap when performing swaps. However, if the deadline has already passed, lots of gas has already been consumed before the transaction fails and this gas will not be refunded to the sender. Adding a simple deadline check at the beginning of the `deposit()` function can prevent this, saving users unnecessary gas costs.

Moreover, this would introduce a `deposit()`-wide deadline check which would be an added feature since the current implementation only applies the deadline to swaps when auto-buy-and-stake is triggered.

Recommendation: Implement a deadline check at the start of the `deposit()` function to prevent execution if the deadline has been violated, ensuring better gas efficiency and broader deadline enforcement.

Resolution: Fixed in e85aa8c

[I-5] Missing/incorrect/misleading natspec

Context: SwapActions.sol#L149-L153, SwapActions.sol#L217-L225, SwapActions.sol#L116, SwapActions.sol#L270-L275, FluxBuyAndBurn.sol#L79, FluxBuyAndBurn.sol#L106

Description: The following functions have incorrect or misleading natspec:

- `SwapActions.getFluxQuoteForInferno` - missing natspec;
- `SwapActions.getInfernoQuoteForTitanX()` - the natspec says that the function gets a quote for Inferno tokens in exchange for Blaze tokens instead of TitanX tokens;
- `SwapActions._sortAmountsForLP()` - the natspec claims that the function creates a Uniswap V3 pool but it only calculates amounts for LP boostrapping;
- `SwapActions._swapInfernoForFlux()` - the natspec says the function swaps Blaze tokens for Inferno tokens but it swaps Inferno tokens for Flux tokens;
- The `FluxBuyAndBurn` constructor natspec says that `Constructor is payable to save gas` which is untrue;
- The natspec for `FluxBuyAndBurn.swapTitanXForFluxAndBurn()` mistakenly says `Swaps TitanX for Inferno and burns the Inferno tokens` while it actually swaps TitanX for Flux and burns the Flux tokens;

Recommendation: Add the natspec where missing or fix it so that it reflects what the actual functions do.

Resolution: Fixed in f55d6f6 and c4f58d6

[I-6] Unused code should be removed

Context: SwapActions.sol#L173-L180, Constants.sol#L61, Constants.sol#L63

Description: The following constants and functions are defined in the codebase but are not used

anywhere. Unused code increases the complexity of the codebase, making it harder to maintain and understand. It can also lead to unnecessary gas costs and deployment size.

- **Constants:**
  - `UNISWAP_V2_FACTORY`
  - `UNISWAP_V2_BLAZE_TITAN_X_POOL`
- **Functions:**
  - `getBlazeQuoteForTitanX()`

Recommendation: Remove all unused constants, functions and other code segments to improve the maintainability and minimize the overall contract size and potential gas costs.

Resolution: Fixed in 8071e8c

[I-7] `FluxAuction.addLiquidityToInfernoBlazePool()` **has a misleading name and NatSpec**

Context: FluxAuction.sol#L157-L161

Description: The function `addLiquidityToInfernoBlazePool()` is named in a way that suggests it adds liquidity to the Inferno/Blaze pool, but it actually bootstraps liquidity for the Flux/Inferno pool.

Additionally, the NatSpec comment for this function states, "Creates a Uniswap V3 pool and adds liquidity," which is misleading. The function only adds liquidity to an existing pool, while the pool itself is created by `Flux._createUniswapV3Pool()`.

Recommendation: 1. Rename the function to `addLiquidityToFluxInfernoPool()` to accurately reflect its purpose. 2. Update the NatSpec comment to correctly describe the function's behavior.

Resolution: Fixed in d9c7b29

[I-8] Lack of event submission

Context: N/A

Description: Almost all of the functions in the contracts are missing emitting events, although they change the state of the contract.

To keep track of historical changes in storage variables, it is recommended that events be emitted for every change in the functions that modify the storage.

Recommendation: Consider emitting events in all functions where state changes to reflect important changes in contracts.

Resolution: Fixed

[I-9] Incorrect event parameter name in `BuyAndBurn` **event**

Context: FluxBuyAndBurn.sol#L66

Description: The `BuyAndBurn` event in the `FluxBuyAndBurn` contract contains a parameter named

`infernoBurnt`, which is misleading. This parameter is intended to track the amount of Flux tokens burnt, not Inferno tokens. The incorrect naming could cause confusion for developers and users who rely on event logs to monitor or analyze protocol behavior.

Recommendation: Rename the `infernoBurnt` parameter in the `BuyAndBurn` event to `fluxBurnt` to accurately reflect the event's purpose and improve code clarity.

Resolution: Fixed in 72b59ce

[I-10] Inconsistent day counting between reward distribution and Flux-to-share ratio in `FluxStaking`

Context: Staking.sol#L242, Staking.sol#L166

Description: There is a minor inconsistency in the `FluxStaking` contract regarding the day-based counting for reward distribution and the Flux-to-share ratio inflation. Specifically:

- Reward distribution is 1-day based (the first day is considered Day #1).
- Flux-to-share ratio inflation is 0-day based (the first day is considered Day #0).

As a result, these two events do not occur on the same day. The Flux-to-share ratio will be updated one day after the 8-day pool reward distribution. While this discrepancy does not have any functional impact on the protocol, it may cause minor confusion and inconsistency in the timing of updates.

Recommendation: Align the day-based counting for both reward distribution and the Flux-to-share ratio inflation to ensure that they occur on the same day, improving consistency and reducing potential confusion.

Resolution: Fixed in 18f4156