Phage
SECURITY

Flare

Security Review

Conducted by:

Pyro, Security Researcher

13.01.2025

Table of Contents

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 reveal the presence of vulnerabilities **but cannot guarantee their absence**.

System overview

Flare is another part of the titanX ecosystem, featuring its buy-and-burn mechanism, but with a twist. Cycles are now weekly, with different days having different distributions. The scope also includes minting and auction contracts, where users can mint Flare.

Executive summary

Overview

Project Name	Flare
Repository	private
Commit hash	e02d2be64dcf3e3e5b6975ba87d24e8019edcc16
Remediation	e02d2be64dcf3e3e5b6975ba87d24e8019edcc16
Methods	Manual review

Timeline

Audit kick-off	08.01.2025
End of audit	12.01.2025
Remediations start	13.01.2025

Scope

src/BuyNBurn/BaseBuyNBurn.sol

src/actions/SwapActions.sol

src/const/Constants.sol

src/core/Flare.sol

src/core/FlareMinting.sol

src/core/FlareBuyNBurn.sol

src/core/FlareAuctionTreasury.sol

src/core/FlareAuction.sol

src/core/FlareAuctionBuy.sol

Issues Found

Severity	Count
High	0
Medium	3
Low	10

Findings

Medium Severity

[M-01] Incorrect distribution in `FlareAuction`

When depositing inside `FlareAuction`, we use the returned amount from `_deposit`, which represents the total amount of `X28` tokens minted from the deposited titanX (76% of the original deposit).

```
IERC20(state.titanX).safeTransferFrom(msg.sender, address(this), _amount);
uint256 X28Amount = _deposit(_amount, false);
_distribute(X28Amount, 0, false);
```

However, `_distribute` later treats this amount as if it were the original titanX deposit. It calculates 76% of that amount and allocates it to BnB and auctionBuy:

```
uint256 _toBuyNBurn = wmul(_amount, TO_BUY_AND_BURN); // 46%
uint256 _toFlareAuctionBuy = wmul(_amount, TO_AUCTION_BUY); // 32%
```

As a result, 24% of the previously minted `X28` tokens remain stuck inside the `FlareAuction` contract. Example:

1. User deposits 100 `titanX`.
2. 76 titanX is converted to X28 tokens, leaving 24 `titanX` unconverted.
3. `_distribute` is called with 76 as _amount, and it sends 46% and 32% of that amount to BnB and auctionBuy, respectively.
4. This leaves 24% (approximately 18 X28) stuck inside the `FlareAuction` contract.

Recommendation:

Replace `X28Amount` with `_amount` in `_distribute` to ensure the correct distribution of funds. Also the below amount needs to be adjusted to match the new allocation, as `_amount` would be 100% of the deposited funds, while the actual swapped X28 would be 76% of that.

```
totalX28Burnt += _amount;
```

Resolution: Fixed

[M-02] `FlareAuction` is insolvent

The `FlareAuction::depositETH` function in the Flare Auction contract tracks two key values: `titanXDeposited` and `titanXAmount`:

```
    uint256 titanXAmount = getSpotPrice(state.WETH, state.titanX, msg.value);
    checkIsDeviationOutOfBounds(state.WETH, state.titanX, msg.value,
       titanXAmount);

    uint256 _genAmount = wmul(msg.value, TO_GENESIS);
    uint256 _swapAmount = msg.value- _genAmount;

    uint256 _titanXToDeposit = _swapWethToTitanX(_swapAmount, _minAmount,
       _deadline);

    uint32 daySinceStart = _daySinceStart();

    UserAuction storage userDeposit = depositOf[msg.sender][++depositId];
    DailyStatistic storage stats = dailyStats[daySinceStart];
    userDeposit.ts = uint32(block.timestamp);
    userDeposit.amount += uint192(titanXAmount);
    userDeposit.day = daySinceStart;

    // Total user deposits > titanXDeposited
    stats.titanXDeposited += uint128(_titanXToDeposit);
```

These values are subsequently used to calculate the Flare token allocation for each user:

```
    function amountToClaim(address _user, uint64 _id) public view returns (
       uint256 toClaim) {
       UserAuction storage userDep = depositOf[_user][_id];
       DailyStatistic memory stats = dailyStats[userDep.day];
    return (userDep.amount * stats.flareEmitted) / stats.titanXDeposited;
}
```

However, there is a discrepancy between `titanXDeposited` and `titanXAmount`. While the user balance is increased by `titanXAmount`, the `titanXDeposited` value is only increased by `_titanXToDeposit`, which is smaller, not only by the 8% we take as fee, but also can be by the 3% that is left as tolerance between the TWAP and instant price.

This results in a significant difference between the deposited and tracked amounts. Consequently, the contract may become insolvent, as early claimers can drain the available Flare tokens, leaving later users with nothing to claim.

Recommendation:

Swap 100% of the ETH sent and then allocate the amounts.

Resolution: Fixed

[M-03] `depositETH` and `mintETH` would brick about 7% of all deposited funds

`_deposit` would swap 84% out (100 - 8%) into X12, which is 77.28%, however `_distribute` would later distribute 76%, leaving 1.28% of the original amount stuck inside the contract as X 28.

```
    // (100 - 8) = 92
    // 92 - (92 * 16) =
77.28
    _deposit(_titanXToDeposit, true);
    _distribute(_titanXToDeposit, _genAmount, true);
```

Even further is that we use the reduced `titanX` amount and `_distribute` sends the old percentages (not adjusted for ETH) example:

1. We deposit 100 titanX worth in ETH
2. 8 of that titanX in ETH gets send to the genesis
3. 92 is swapped - _titanXToDeposit == 92
4. _deposit removes 16% and mints X28 with the rest X12 - 77.28 and titanX - 14.72
5. _distribute distributes 48% and 28% of the 92 - 44.16 and 25.76 (69.92 X28 in total)
6. _distribute also sends 2x8% of the titanX around –> `2 * 7.36 = 14.72`

From what we can see due to the reduced value of titanX, and the default percentages being used we would incur a loss of `77.28 - 69.92 = 7.36` X12, which is 7.36% of our original amount left stuck inside the contract.

Recommendation:

Consider swapping the whole amount to `titanX` and using it the same way as in mint/deposit.

Resolution: Fixed

Low Severity

[L-01] `_distribute` distributes less `titanX`

During the minting process, before _distribute is called, we enter _deposit, which converts 76% of the deposited titanX into X 28. This means that after the deposit, only 24% of the funds remain in titanX.

```
    _amount -= wmul(_amount, TITANX_TO_DEDUCT); //
0.24e18 -> 24%
    mintingState.titanX.approve(address(mintingState.X28), _amount);
    uint256 _X28BalBefore = mintingState.X
28.balanceOf(address(this));
    mintingState.X
28.mintX28withTitanX(_amount);
    uint256 _X28BalAfter = mintingState.X
28.balanceOf(address(this));
```

The `_distribute` function, which is later used to distribute the deposited titanX, mistakenly distributes less than intended. This happens because it uses titanXBalance instead of _amount. As a result, it distributes 24% of the total titanX balance (2x8% allocations for FlARE and 1x8% for genesis) instead of distributing the intended amount from _amount.

```
   function _distribute(uint256 _amount, uint256 _ethGenAmount,
      bool _isEth) internal {
      uint256 titanXBalance = mintingState.titanX.balanceOf(address(this));
      uint256 X28Balance = mintingState.X
28.balanceOf(address(this));
      if (_amount == 0) return;
         _distributeGenesis(_ethGenAmount, titanXBalance, _isEth); // 8%
         uint256 _toTitanXInfBnB = wmul(titanXBalance, TO_INFERNO_BNB); // 8%
         if (block.timestamp ⩽ startTimestamp + FOUR_WEEKS) {
         mintingState.titanX.transfer(FlARE_LP, wmul(titanXBalance,
            TO_FLARE_LP)); //8%
      } else {
         mintingState.titanX.transfer(mintingState.titanXInfBnB,
            _toTitanXInfBnB); //8%
      }

      mintingState.titanX.transfer(FlARE_LP, wmul(titanXBalance,
         TO_FLARE_LP)); // 8%
   }
```

Example:

1. User mints with 100e18 titanX.
2. 76e18 gets mined to X28, leaving 24e18 in titanX.
3. `_distribute` incorrectly calculates the 8% allocations from the total titanX balance instead of the `_amount`, resulting in 1x 1.92e18 sent to genesis and 2x 1.92e18 transfers to `FlARE_LP`.

Note that the same is done inside FlareAuction too.

Recommendation:

Replace `titanXBalance` with `_amount` in `_distribute` to ensure the correct distribution amount.

Resolution: Fixed

[L-02] `_distribute` distributes too much X28

During the minting process, _deposit converts 76% of the deposited titanX into X28:

```
   _amount -= wmul(_amount, TITANX_TO_DEDUCT); //
0.24e18 -> 24%
   mintingState.titanX.approve(address(mintingState.X28), _amount);

   uint256 _X28BalBefore = mintingState.X
28.balanceOf(address(this));

   mintingState.X
28.mintX28withTitanX(_amount);
   uint256 _X28BalAfter = mintingState.X
28.balanceOf(address(this));
```

However, later in the process, we attempt to distribute 78% of `_amount` (46% for burning and 32% for the FlARE auction), even though only 76% was converted to X 28.

```
   uint256 _toBuyNBurn = wmul(_amount, TO_BUY_AND_BURN); // 46%
   uint256 _toFlareAuctionBuy = wmul(_amount, TO_AUCTION_BUY); // 32%

   mintingState.X
28.approve(address(mintingState.bnb), _toBuyNBurn);
   mintingState.bnb.distributeX28ForBurning(_toBuyNBurn);
   mintingState.X
28.approve(mintingState.flare.FlareAuctionBuy(), _toFlareAuctionBuy);

   FlareAuctionBuy(mintingState.flare.FlareAuctionBuy()).distribute(
      _toFlareAuctionBuy);
```

However, we try to distribute 78% of `_amount` in X28 allocations, but only 76% is available.

Recommendation:

The documentation specifies that these values should be 48% and 28%, respectively. Consider adjusting the percentages to match the available 76%, or reduce them by 2% to ensure proper distribution.

Resolution: Fixed

[L-03] `FlareMinting::mintETH` is susceptible to price manipulation

The `FlareMinting::mintETH` function currently relies on the spot price to determine the amount of titanX required for flare minting:

```
    uint256 titanXAmount = getSpotPrice(address(mintingState.WETH), address(
        mintingState.titanX), msg.value);
    checkIsDeviationOutOfBounds(address(mintingState.WETH), address(
        mintingState.titanX), msg.value, titanXAmount);

    // ...
    uint256 flareAmount = (titanXAmount * getRatioForCycle(currentCycle)) /
        1e18;
```

This dependency on the spot price introduces a vulnerability where the spot price can be manipulated. The `checkIsDeviationOutOfBounds` function permits up to a 3% deviation between the TWAP and the current spot price. This gap can be exploited by malicious actors through MEV attacks to siphon funds.

Additionally, with the TWAP window set between 5 to 30 minutes and the 3% deviation cap, the mintETH function may frequently revert during normal market fluctuations, causing usability issues.

This issue is also present in the `FlareAuction` function.

Recommendation:

To mitigate this vulnerability, modify the mintETH function to swap the entire ETH input to titanX and proceed as with a normal deposit. This adjustment would eliminate the need for deviation checks, preventing mintETH from reverting during regular market conditions and closing the MEV exploitation gap.

Resolution: Acknowledged

[L-04] `_distribute` fails to properly allocate titanX

The _distribute function is unable to properly distribute when `amountLeft ⩾ _amount`, as 100% of `_amount` is being sent to `FlARE_LP`:

```
    if (totalSentToLP < INITIAL_X28_SENT_TO_LP) { // 10e27
        uint256 amountLeft = INITIAL_X28_SENT_TO_LP - totalSentToLP;

        uint256 amountToAdd = amountLeft ⩾ _amount
        ? _amount
        : amountLeft;

        totalSentToLP += amountToAdd;
        _amount -= amountToAdd;
        mintingState.X
28.transfer(FlARE_LP, amountToAdd);
}
```

The issue occurs because `_amount` represents the titanX deposited, while we only mint 76% of the deposited amount in X28 tokens. Consequently, transferring the full `_amount` to `FlARE_LP` exceeds the available minted X 28.

Example: 1. Depositing 100 titanX results in minting only 76 X28 tokens. 2. _distribute allocates the entire 100 `_amount` to `FlARE_LP`, which is incorrect since only 76 X28 tokens were minted.

Recommendation:

Modify the `_distribute` logic to swap the remaining titanX for X28 and allocate it accordingly when entering the if condition. This ensures that only the minted X28 tokens are distributed and prevents over-allocation.

Resolution: Fixed

[L-05] Discrepancy in _distribute

Both auction and mint contracts have _distribute. However, there is a difference in how both of them implement it. FlareMinting distributes 8% to `FlARE_LP` for the first 4 weeks, as it's well described in the docs.

```
if (block.timestamp ⩽ startTimestamp + FOUR_WEEKS) {
   mintingState.titanX.transfer(FlARE_LP, wmul(titanXBalance,
      TO_FLARE_LP)); // 8%
} else {
   mintingState.titanX.transfer(mintingState.titanXInfBnB,
      _toTitanXInfBnB); // 8%
}
mintingState.titanX.transfer(FlARE_LP, wmul(titanXBalance,
   TO_FLARE_LP)); // 8%
```

However, `FlareAuction` doesn't follow this logic and directly sends 8% to BnB and 8% to `Flare_LP`.

```
// 8% of the TITANX balance to inferno pool
uint256 _toTitanXInfBnB = wmul(titanXBalance, TO_INFERNO_BNB);
IERC20(state.titanX).transfer(state.titanXInfBnB, _toTitanXInfBnB);
// 8% of the TITANX balance to the LP
IERC20(state.titanX).transfer(FlARE_LP, wmul(titanXBalance, TO_FLARE_LP));
```

Recommendation:

Add the same if condition inside FlareAuction to ensure that it distributes the funds correctly based on the initial 4-week period.

Resolution: Fixed

[L-06] `toGenesis` is one order of magnitude bigger

The docs explain that after the minting phase the buy/sell tax should be 1% for the daily auction pool, 1% for burn, and 0.8% to genesis. However, in the current implementation, the amount sent to genesis is incorrectly set to 8% instead of 0.8%.

```
   toGenesis = currentCycle >= MAX_MINT_CYCLE
      ? wmul(value, AFTER_MINT_BUY_SELL_TAX_GENESIS) //
0.08e18 -> 8%
      : wmul(value, MINT_BUY_SELL_TAX_GENESIS); //
0.01e18 -> 1%
```

Recommendation:

Lower `AFTER_MINT_BUY_SELL_TAX_GENESIS` to 0.008e18 to correct the tax rate to 0.8%.

Resolution: Fixed

[L-07] `swapX28ToflareAndFeedTheAuction` can be MEVed

The current `swapX28ToflareAndFeedTheAuction` can be exploited for MEV if privateMode is disabled, as it is currently initialized. This vulnerability allows users to config- ure extremely low slippage values, making the trade susceptible to sandwich attacks.

```
   uint256 prevFlareBalance = flare.balanceOf(address(this));
   _swapX28ForFlare(currInterval.amountAllocated - incentive, _amountFlareMin,
      _deadline);
   uint256 currFlareBalance = flare.balanceOf(address(this));
```

This same risk is present in the FlareBuyNBurn function.

Recommendation:

Consider implementing TWAP checks to handle slippage. Alternatively, set `privateMode` to true in the constructor and ensure it remains enabled. The same recommendation applies to the `FlareBuyNBurn` function.

Resolution: Fixed

[L-08] Incorrect allocation during the last cycle in emitForAuction

The intended allocation behavior as per documentation (https://flare-4.gitbook.io/flare/mint-allocation-and-ratio) should be 11% of the pool daily during the minting phase, specifically up to and including the 11th cycle. Post-minting phase, from the

12th cycle onward, the allocation should switch to 5% daily. However, the current condition incorrectly applies the 5% allocation starting in the 11th cycle instead of after it.

The issue stems from the conditional logic:

```
   uint256 auctionAllocation = currentCycle ⩾ MAX_MINT_CYCLE ?
0.05e18 :
0.11e18;
```

Additionally, `getCurrentMintCycle` does not differentiate between the 11th cycle and subsequent cycles, causing this misalignment throughout the codebase.

Recommendation:

Modify the condition to apply the 11% allocation correctly during the 11th cycle and switch to 5% from the 12th cycle onward. Update the logic as follows: - Ensure `getCurrentMintCycle` returns 12 after the 11th cycle. - Adjust the condition in `FlareAuctionTreasury` to use `currentCycle > MAX_MINT_CYCLE`. - Fix the related logic inside `Flare::_update` to reflect the correct allocation transition timing.

Resolution: Fixed

[L-09] startTimestamp can be any day, but it should be constrained to only Friday

Docs describe how minting should start every Friday 2pm UTC and last 24h. However, minting is started right after startTimestamp, meaning that in order for the code to work right, startTimestamp must be on a Friday.

```
function getCurrentMintCycle() public view returns (uint32 currentCycle,
   uint32 startsAt, uint32 endsAt) {
   uint32 timeElapsedSince = uint32(block.timestamp - startTimestamp);
   currentCycle = uint32(timeElapsedSince / GAP_BETWEEN_CYCLE) + 1; // 7
      days

   if (currentCycle > MAX_MINT_CYCLE) currentCycle = MAX_MINT_CYCLE; // 11
   startsAt = startTimestamp + ((currentCycle - 1) * GAP_BETWEEN_CYCLE);
   // Cycles start at first day on 'startTimestamp', i.e. if
   // 'startTimestamp' is Monday 2pm UTC
   // minting will be 'mon 2pm - tue 2pm'
   endsAt = startsAt + MINT_CYCLE_DURATION; // 24h
}
```

Recommendation:

When setting `startTimestamp`, make sure it's on a Friday.

Resolution: Acknowledged

[L-10] `_calculateIntervals` can miss interval distributions when turning to the next week

BaseBuyNBurn::_calculateIntervals will sometimes miss one or more interval distributions. These funds will not be locked or stuck, but instead would be distributed in later intervals. This occurs in the second iteration of the while loop inside forAllocation and more specifically in the balanceOf >= check.

```
uint256 forAllocation = lastSnapshotTimestamp + 1 weeks > end
   ? totalX28Distributed
   : balanceOf ⩾ _totalAmountForInterval +
      wmul(diff,getDailyTokenAllocation(end))
      ? diff
      : 0;
```

If we have just switched to a new week, and balanceOf is:

1. Bigger than pending intervals

```
(1,2 or 10 intervals) - _totalAmountForInterval + (diff *
   getDailyTokenAllocation(end)* accumulatedIntervalsForTheDay /
   INTERVALS_PER_DAY)
```

2. Smaller than all intervals for the day

```
_totalAmountForInterval + (diff * getDailyTokenAllocation(end))
```

Then the check would return 0 and we won't distribute for the few intervals that we have already passed.

Recommendation:

Consider including `accumulatedIntervalsForTheDay` math into the check to verify if we have enough balance to distribute for the few intervals we have passed for the new day.

Resolution: Fixed