e

CD SECURITY

AUDIT REPORT

Phoenix
November 2024

Prepared by
hals
0xnevi

Introduction

A time-boxed security review of the **Phoenix** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Phoenix**

The Phoenix protocol ecosystem is centered around `$Phoenix` tokens, incorporating mechanisms for staking, auctions, minting, and controlling supply. Users can participate in daily auctions with `$TitanX` to acquire `$Phoenix` tokens, and they can also mint `$Phoenix` tokens by depositing `$TitanX` during defined cycles.

Staking contracts receive portions of the deposits from the `Minting` and `Auction` contracts and implement mechanisms to swap these tokens and stake them. This involves staking `$Blaze` & `$TitanX` tokens to earn ETH rewards, and stake `$Flux` tokens to earn `$TitanX` rewards, which are reinvested into the system to buy more tokens for further staking.

The protocol integrates Uniswap V3 for liquidity and swaps, automates a token buy-and-burn process to reduce the `$Phoenix` supply, and fuels the auction contract with `$Phoenix`. 50% of the `$Phoenix` tokens bought are transferred to the auction contract, while the remaining `$Phoenix` tokens are burned. This approach helps maintain token supply and incentivizes ecosystem growth.

Severity classification

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

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 96446acb3c9c6e5479875aba045347c33d6faf70**

Scope

The following smart contracts were in scope of the audit:

- `src/actions/*`
- `src/const/*`
- `src/interfaces/*`
- `src/staking/*`
- `src/Auction.sol`
- `src/AuctionTreasury.sol`
- `src/BuyAndBurn.sol`
- `src/Minting.sol`
- `src/Phoenix.sol`
- `src/TitanXStakingManager.sol.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 4 issues
- Low: 5 issues

Findings Summary

ID	Title	Severity	Status
[M-01]	`stake()` functions calculate the incentive based on the token balance not with the amount to be staked	Medium	Fixed
[M-02]	`Minting._sortAmountsForLP()` implements a high slippage percentages on the minimum accepted tokens of the liquidity position	Medium	Fixed
[M-03]	`SwapAction.getTwapAmountV3()`: the implemented fallback mechanism for `secondsAgo` is vulnerable to price manipulation	Medium	Fixed
[M-04]	Loss of phoenix fee collections from initial liquidity	Medium	Acknowledged
[L-01]	`BlazeStakingVault.stakeBlaze()` : incorrect check for cooldown and minimum stake amount	Low	Fixed
[L-02]	Dust amounts deposited can allow claim without deposits	Low	Acknowledged
[L-03]	Unnecessary delay in phoenix token claims	Low	Acknowledged

ID	Title	Severity	Status
[L-04]	`depositId` overflow can cause funds in `AuctionTreasury` to be stuck	Low	Fixed
[L-05]	Consider allowing flexible slippage for swap actions	Low	Acknowledged

Detailed Findings

[M-01] `stake()` functions calculate the incentive based on the token balance not with the amount to be staked

Severity

Impact: Medium

Likelihood: High

Description

- `stake()` function in the three staking vaults calculates the incentive based on the total balance of the tokens instead of the actual balance that is going to be staked.
- For example: in `BlazeStakingVault.stakeBlaze()`: the incentive is calculated based on the availabe blaze balance, then this incentive is deducted from the available balance to get the `blazeToStake`, then this value is checked to be greater than `minStakeAmount` and **less** than `maxStakeAmount`, so if it's less than the maximum limit; `blazeToStake` is updated to equal that maximum limit:

```
function stakeBlaze() external onlyByOwnerInPrivateMode {
        State storage _state = state;

        uint256 blazeToStake = blaze.balanceOf(_this());

        uint256 incentive = wmul(blazeToStake, state.incentive);
        blazeToStake -= incentive;

        require(_state.lastStakeTs != 0 || blazeToStake >=
_state.minStakeAmount, CooldownNotPassed());
        require(
            block.timestamp - _state.lastStakeTs >= _state.stakingCooldown
|| blazeToStake >= _state.minStakeAmount,
            CooldownNotPassed()
        );

        if (blazeToStake > _state.maxStakeAmount) blazeToStake =
```

```
_state.maxStakeAmount;

        blazeStaking.stakeBlaze(blazeToStake, BLAZE_MAX_STAKE);

        blaze.transfer(msg.sender, incentive);

        emit Staked(++_state.lastStakingPosition, blazeToStake);

        _state.lastStakeTs = uint32(block.timestamp);
    }
```

- So as can be noticed, the incentive sent to the user for calling `blazeStake()` can be much larger than intended **if the initial `blazeToStake` is greater than `maxStakeAmount`** as the incentive is calculated based on the total balance and not based on the updated `blazeToStake` amount; resulting in sending the user a large incentive than intended.
- Same issue in `FluxStakingVault.stake()` & `TitanXStakingVault.stake()` functions.

Recommendations

Calculate the incentive based on the final `blazeToStake` that's going to be staked :

```
function stakeBlaze() external onlyByOwnerInPrivateMode {
        State storage _state = state;

        uint256 blazeToStake = blaze.balanceOf(_this());

-       uint256 incentive = wmul(blazeToStake, state.incentive);
-       blazeToStake -= incentive;

        require(_state.lastStakeTs != 0 || blazeToStake >=
_state.minStakeAmount, CooldownNotPassed());

        require(
            block.timestamp - _state.lastStakeTs >= _state.stakingCooldown
|| blazeToStake >= _state.minStakeAmount,
            CooldownNotPassed()
        );

        if (blazeToStake > _state.maxStakeAmount) blazeToStake =
_state.maxStakeAmount;


+       uint256 incentive = wmul(blazeToStake, state.incentive);
+       blazeToStake -= incentive;
+       require(blazeToStake >= _state.minStakeAmount);

        blazeStaking.stakeBlaze(blazeToStake, BLAZE_MAX_STAKE);

        blaze.transfer(msg.sender, incentive);
```

```
        emit Staked(++_state.lastStakingPosition, blazeToStake);

        _state.lastStakeTs = uint32(block.timestamp);
    }
```

[M-02] `Minting._sortAmountsForLP()` implements a high slippage percentages on the minimum accepted tokens of the liquidity position

Severity

Impact: Medium

Likelihood: Medium

Description

- `Minting._sortAmountsForLP()` is called to sort tokens and calculate the accepted minimum amounts when the owner of the contract adds liquidity to the inferno-phoenix Uni-v3 pool via `Minting.addLiquidityToInfernoPhoenixPool()`:

```
function _sortAmountsForLP(uint256 _infernoAmount, uint256 _phoenixAmount)
        internal
        view
        returns (
            uint256 amount0,
            uint256 amount1,
            uint256 amount0Min,
            uint256 amount1Min,
            address token0,
            address token1
        )
    {
        address _phoenix = address(phoenix);
        address _inferno = address(inferno);

        (token0, token1) = _phoenix < _inferno ? (_phoenix, _inferno) :
(_inferno, _phoenix);
        (amount0, amount1) = token0 == _phoenix ? (_phoenixAmount,
_infernoAmount) : (_infernoAmount, _phoenixAmount);

        (amount0Min, amount1Min) = (wmul(amount0, uint256(0.2e18)),
wmul(amount1, uint256(0.2e18)));
    }
```

- As can be noticed, the minimum amounts calculated as 20% of the amounts that are going to be added as a liquidity, which means accepting a 80% slippage.

- In case of market volatility during liquidity addition; this would result in accepting low amounts of tokens for the created position, which will result in collecting less fees (`$Phoenix` & `$Inferno`) for that position; thus affecting the amounts of `$Inferno` tokens that are going to be sent to the `fluxStakingVault` for staking (after swapping to `$Flux`).

Recommendations

```
function _sortAmountsForLP(uint256 _infernoAmount, uint256 _phoenixAmount)
        internal
        view
        returns (
            uint256 amount0,
            uint256 amount1,
            uint256 amount0Min,
            uint256 amount1Min,
            address token0,
            address token1
        )
    {
     //...

-       (amount0Min, amount1Min) = (wmul(amount0, uint256(0.2e18)),
wmul(amount1, uint256(0.2e18)));

+       (amount0Min, amount1Min) = (wmul(amount0, uint256(0.8e18)),
wmul(amount1, uint256(0.8e18)));
    }
```

[M-03] `SwapAction.getTwapAmountV3()`: the implemented fallback mechanism for `secondsAgo` is vulnerable to price manipulation

Severity

Impact: Medium

Likelihood: Medium

Description

`SwapAction.getTwapAmountV3()` function is called whenever a swap action is invoked via `swapExactInputV3()`, where it uses twap to ensure the swap is performed within the slippage tolerance:

```
function getTwapAmountV3(address tokenIn, address tokenOut, uint256
amount)
        public
```

```
        view
        returns (uint256 twapAmount, uint224 slippage)
    {
        address poolAddress = PoolAddress.computeAddress(v3Factory,
PoolAddress.getPoolKey(tokenIn, tokenOut, POOL_FEE));

        Slippage memory slippageConfig = slippageConfigs[poolAddress];

        if (slippageConfig.twapLookback == 0 && slippageConfig.slippage ==
0) {
            slippageConfig = Slippage({twapLookback: 15, slippage: WAD -
0.2e18});
        }

        uint32 secondsAgo = slippageConfig.twapLookback * 60;

        uint32 oldestObservation =
OracleLibrary.getOldestObservationSecondsAgo(poolAddress);

        if (oldestObservation < secondsAgo) secondsAgo =
oldestObservation;

        (int24 arithmeticMeanTick,) = OracleLibrary.consult(poolAddress,
secondsAgo);

        uint160 sqrtPriceX96 =
TickMath.getSqrtRatioAtTick(arithmeticMeanTick);

        slippage = slippageConfig.slippage;

        twapAmount = OracleLibrary.getQuoteForSqrtRatioX96(sqrtPriceX96,
amount, tokenIn, tokenOut);
    }
```

where the `oldestObservation` represents the timestamp difference between the oldest recorded observation and the current time.

- If `secondsAgo` (the TWAP window) exceeds `oldestObservation`, the function defaults `secondsAgo` to `oldestObservation` as a fallback mechanism instead of reverting:

```
if (oldestObservation < secondsAgo) secondsAgo = oldestObservation;
```

- But `oldestObservation < secondsAgo` means that the pool doesn't have enough historical data (low cardinality), and the available data will be used regardless of its correctness.
- In UNI-V3, the oldest observation can be updated when the first trade occures in a new block, and if the pool has low cardinality (initialized to 1 for example) and the fallback mechanism uses `oldestObservation` as `secondsAgo`, then any malicious actor can frontrun the `swapExactInputV3()` with a malicious trade before the TWAP is calculated to manipulate the price

(inflate it) which will result in returning low calculated `twapAmount` that wouldn't protect against slippage.

Recommendations

- revert the txn if `oldestObservation < secondsAgo`.
- ensure that the used pools have a sufficient number of cardinality.
- use Chainlink oracles as a fallback in case twap fails.

[M-04] Loss of phoenix fee collections from initial liquidity

Impact: Medium

Likelihood: Medium

Description

In `Minting.addLiquidityToInfernoPhoenixPool`, it allows the admin to invoke a one-time initial liquidity addition of `INITIAL_TITAN_X_FOR_LIQ` to the INF/PHONIEX pool, with a subsequent position minted to the Minting contract.

The fees obtained for contributing to the initial liquidity of the INF/PHOENIX pool can be collected via an admin only `collectFees` function. Notice how the obtained inferno is transferred to the `fluxStakingVault` to be staked, whereas the `phoenix` obtained is simply burned, essentially representing a loss of fees.

```
    function collectFees() external returns (uint256 amount0, uint256
amount1) {
        LP memory _lp = lp;

        INonfungiblePositionManager.CollectParams memory params =
INonfungiblePositionManager.CollectParams({
            tokenId: _lp.tokenId,
            recipient: address(this),
            amount0Max: type(uint128).max,
            amount1Max: type(uint128).max
        });

        (amount0, amount1) =
INonfungiblePositionManager(positionManager).collect(params);

        (uint256 phoenixAmount, uint256 infernoAmount) =
_lp.isPhoenixToken0 ? (amount0, amount1) : (amount1, amount0);

@>      phoenix.burn(phoenixAmount);
        inferno.transfer(address(fluxStakingVault), infernoAmount);
    }
```

Recommendation

Consider a 50:50 split similar to the `BuyAndBurn` contract, where 50% is burned and 50% is transferred to the auctionTreasury to be auctioned off to provide fuel for the daily auctions which in turn provides value back to the buy&bid / buy&burn.

[L-01] `BlazeStakingVault.stakeBlaze()` : incorrect check for cooldown **and** minimum stake amount

Description

`stakeBlaze()` incorrectly checks for the minimum amount and cool down period, as it allows staking if the cooldown period passed **or** if the `blazeToStake` amount is greater than the `minStakeAmount`,
but the function Natspec clearly states that **both** conditions should be met before staking:

```
    /**
     * @notice Stakes all accumulated Blaze tokens in the Blaze staking
contract.
     * @dev Requires the cooldown period to have passed since the last
stake and that the balance
     * of Blaze tokens is above the threshold for the first stake. This
function increments
     * the staking position counter upon successful staking.
     */
 function stakeBlaze() external onlyByOwnerInPrivateMode {
    //...
            require(
                    _state.lastStakeTs != 0 || blazeToStake >=
_state.minStakeAmount,
                    CooldownNotPassed()
                );

            require(
            block.timestamp - _state.lastStakeTs >= _state.stakingCooldown
|| blazeToStake >= _state.minStakeAmount,
            CooldownNotPassed()
        );
    //...
 }
```

- Same issue in `FluxStakingVault.stake()` & `TitanXStakingVault.stake()` functions.

Recommendations

Check for the cooldown period **and** minimum amount to be staked:

```
 function stakeBlaze() external onlyByOwnerInPrivateMode {
    //...
            require(
-           block.timestamp - _state.lastStakeTs >= _state.stakingCooldown
|| blazeToStake >= _state.minStakeAmount,
+           block.timestamp - _state.lastStakeTs >= _state.stakingCooldown
&& blazeToStake >= _state.minStakeAmount,
            CooldownNotPassed()
        );
    //...
 }
```

[L-02] Dust amounts deposited can allow claim without deposits

Description

In `Minting.mint`, the user deposits `amount` worth of TitanX tokens to eventually mint phoenix tokens depending on which period of the 28 day cycle he deposited in. The distribution of these tokens are performed through the `_distribute` function.

```
    function _distribute(uint256 _amount) internal {
        uint256 titanXBalance = titanX.balanceOf(address(this));
        // @note - If there is no added liquidity, but the balance exceeds
the initial for liquidity, we should distribute the difference
        if (!lp.hasLP) {
            if (titanXBalance <= INITIAL_TITAN_X_FOR_LIQ) return;
            _amount = uint192(titanXBalance - INITIAL_TITAN_X_FOR_LIQ);
        }

        titanX.transfer(address(fluxStakingVault), wmul(_amount,
uint256(0.28e18)));
        titanX.transfer(titanXVault, wmul(_amount, uint256(0.2e18)));
        titanX.transfer(address(buyAndBurn), wmul(_amount,
uint256(0.35e18)));
        titanX.transfer(address(blazeStakingVault), wmul(_amount,
uint256(0.09e18)));
        titanX.transfer(GENESIS, wmul(_amount, TO_GENESIS));
    }
```

Notice even though all values are scaled to 18 decimals, if a small enough `amount` value is utilized, it can cause a round down (e.g, 2 wei would work here), essentially not distributing any tokens to the relevant addresses.

Note

- The same issue exists in `Auction`, although a smaller value such as 1 wei needs to be utilized

- This issue is set as low severity given this contracts are expected to be integrated on mainnet and gas costs would likely not incentivize this behavior.

Recommendation

Consider a minimum deposit amount (e.g. 100 wei), this will immediately disallow any sort of rounding down to zero

[L-03] Unnecessary delay in phoenix token claims

Description

In `Minting.mint`, the delay is fixed at 24 hours after the time of deposit. This essentially means that early depositors of a specific cycle can claim faster than later depositors (up to a maximum of 24 hours apart)

```
    function claim(uint96 _depositId) public {
        UserDeposit memory userDep = userDeposit[msg.sender][_depositId];

@>      require(block.timestamp > userDep.depositedAt + 24 hours,
CycleStillOngoing());

        (uint32 cycle,,) = getCycleAt(userDep.depositedAt);

        uint256 toClaim = wmul(userDep.amount, getRatioForCycle(cycle));

        delete userDeposit[msg.sender][_depositId];

        emit ClaimExecuted(msg.sender, toClaim, _depositId);

        totalPhoenixClaimed = totalPhoenixClaimed + toClaim;

        phoenix.mint(msg.sender, toClaim);
    }
```

However, this is not necessary since any deposit before the end of the cycle is still a legitimate deposit, and the minting ratio would be fixed to that specific cycle.

Recommendation

Consider utilizing the following check instead

```
    function claim(uint96 _depositId) public {
        UserDeposit memory userDep = userDeposit[msg.sender][_depositId];
+       require(getCycleAt(block.timestamp) > getCycleAt(depositedAt),
CycleStillOngoing());
-       require(block.timestamp > userDep.depositedAt + 24 hours,
CycleStillOngoing());
```

```
        (uint32 cycle,,) = getCycleAt(userDep.depositedAt);

        uint256 toClaim = wmul(userDep.amount, getRatioForCycle(cycle));

        delete userDeposit[msg.sender][_depositId];

        emit ClaimExecuted(msg.sender, toClaim, _depositId);

        totalPhoenixClaimed = totalPhoenixClaimed + toClaim;

        phoenix.mint(msg.sender, toClaim);
    }
```

[L-04] `depositId` overflow can cause funds in `AuctionTreasury` to be stuck

Description

In `Auction.deposit`, each time a users deposit a global `depositId` is incremented and assigned to the user.

```
    function deposit(uint192 _amount) external notAmount0(_amount) {
        require(block.timestamp >= startTimestamp,
PhoenixAuction__NotStartedYet());

        _updateAuction();

        uint32 daySinceStart = _daySinceStart();

@>      UserAuction storage userDeposit = depositOf[msg.sender]
[++depositId];

        DailyStatistic storage stats = dailyStats[daySinceStart];

        userDeposit.ts = uint32(block.timestamp);
        userDeposit.amount = _amount;
        userDeposit.day = daySinceStart;

        stats.titanXDeposited += uint128(_amount);

        _distribute(_amount);

        emit UserDeposit(msg.sender, _amount, daySinceStart);
    }
```

Notice that this is also the only function where the internal `_updateAuction` is called, where the daily phoenix (currently at 1%), is emitted from the `AuctionTresury` to the `Auction` contract. If a malicious user cause an overflow in `depositId` by performing huge amounts of 1 wei deposits (since TitanX is a 18

decimal token, it would only require 19 tokens), this can lead to permanently stuck funds within the `AuctionTreasury` contract since there is no other way to pull funds out.

Note: Since this contract is likely deployed on mainnet, gas costs will likely disincentivize this attack. However, if future cheaper L2 chains is to be integrated, this could be of concern.

Recommendation

Consider:

- Declaring the global `depositId` to `uint256`

[L-05] Consider allowing flexible slippage for swap actions

Description

In all contracts that inherits the abstract contract `SwapActions`, the slippage is a fixed parameter set by the admin via `changeSlippageConfig` within the `slippageConfigs` mapping variable.

```
    function swapExactInputV3(address tokenIn, address tokenOut, uint256
tokenInAmount, uint32 deadline)
        internal
        returns (uint256 amountReceived)
    {
        (uint256 twapAmount, uint224 slippage) = getTwapAmountV3(tokenIn,
tokenOut, tokenInAmount);

        IERC20(tokenIn).approve(v3Router, tokenInAmount);

        ISwapRouter.ExactInputParams memory params =
ISwapRouter.ExactInputParams({
            path: abi.encodePacked(tokenIn, POOL_FEE, tokenOut),
            recipient: address(this),
            deadline: deadline,
            amountIn: tokenInAmount,
            amountOutMinimum: wmul(twapAmount, slippage)
        });

        return ISwapRouter(v3Router).exactInput(params);
    }
```

This means that if a slippage adjustment is intended, even during times of high price volatility, it would take a two step change of invoking `changeSlippageConfig` first before invoking any intended actions, which can involve higher gas costs and user experience. This impacts the following functions:

- `BuyAndBurn.buyNBurn`
- `Minting.addLiquidityToInfernoPhoenixPool`

- `TitanXStakingVault.buyTitanX`
- `FluxStakingVault.buyinferno`
- `FluxStakingVault.buyFlux`
- `BlazeStakingVault.buyinferno`
- `BlazeStakingVault.buyBlaze`

Recommendation

Consider allowing a direct slippage input within the relevant swap actions.