Blaze Security Review

Version 1.0

24.06.2024

Conducted by: nmirchev8 , SR
deth , SR

Table of Contents

1 About Egis Security

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

Both members of Egis Security have a proven track record on public auditing platforms such as Code4rena, Sherlock & Codehawks, uncovering more than 100 High/Medium severity vulnerabilities, with multiple first and top place finishes.

2 Disclaimer

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

3 Risk classification

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

3.1 Impact

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

3.2 Likelihood

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

3.3 Actions required by severity level

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

4 Executive summary

Overview

Project Name	Blaze
Repository	https://github.com/ShintoSan/blaze-contract/tree/audit-final
Commit hash	9f6a3522fcbdcada82be462acdae15651ca766ee
Documentation	https://docs.titanblaze.win/
Methods	Manual review

Scope

contracts/BlazeBonfire.sol
contracts/BlazeBurner.sol
contracts/BlazeBuyAndBurn.sol
contracts/blazeStaking.sol
contracts/DiamondHand.sol

Issues Found

Critical risk	1
High risk	2
Medium risk	4
Low risk	10
Informational	10

5 Findings

5.1 Critical risk

5.1.1 `BlazeBurner.sol#getTitanQuoteForEth()` - The function always uses `slot0` to calculate slippage

Severity: *Critical risk*

Context: BlazeBurner.sol#L31

Description: Based on the value of `secondsAgo` the protocol will use either `slot0` or twap in order to quote Titan for ETH. `secondsAgo` is first set here.

```
uint32 secondsAgo = _titanPriceTwa * 60;
```

The issue is that `_titanPriceTwa` has no setter function, so the value will always be 0, meaning that `secondsAgo` will always be 0.

Everytime the function is called, `slot0` will be used, which is vulnerable to sandwich attacks which will result in a loss of funds for the protocol.

Recommendation: Add a setter function for `_titanPriceTwa` or use BlazeBuyAndBurn's `_titanPriceTwa`, like it's done in `getSlippage`.

Resolution: Fixed

5.2 High risk

5.2.1 `DiamondHand.sol#getDiamontNFT()` - One NFT can be used to game the ticket system

Severity: *High risk*

Context: DiamonHand.sol#L98-L100

Description: When a user wants to participate in the diamond hand contract, he calls `participate`. The function calculates tickets based of 3 functions: `getMintedNFT`, `getDiamonNFT` and `getBlazeStake`.

`getDiamondNFT` calls `DiamonHandWrapper.sol#getDiamondNFT` which then calls `DiamondNFT.sol#userDiamondNFT`

```
mapping(address => uint) public userDiamondNFT;
```

This is just a mapping in the NFT contract. The mapping stores special "diamond" NFT's which can be received when minting the NFT. The issue here is that, these NFT's can be freely transferred at any time. This opens up the DiamonHand to a sybil style attack. Example: 1. Alice has 1 Diamond NFT. 2. She calls participate and adds 1 ticket to her address. 3. Alice then transfers the NFT to another one of her addresses and calls participate again. 4. Alice now has 2 tickets although she used 1 NFT.

The attack is infinitely repeatable and will heavily inflate all the tickets in a cycle and will give users unfair amount of rewards based on a single NFT.

Recommendation: The best fix would be to transfer the NFT when participate is called.

Resolution: Fixed

5.2.2 `BlazeBuyAndBurn.sol#_swapTitanToBlaze()` - `adjustedBlazeAmount` is scaled incorrectly

Severity: *High risk*

Context: BlazeBuyAndBurn.sol#L428

Description: **NOTE** PLEASE CAREFULLY TEST RECOMMENDED FIX

`titanPriceInBlaze` retrieves the value of TitanX in Blaze.

Then `adjustedBlazeAmount` is calculated like so:

```
uint256 adjustedBlazeAmount = ((titanPriceInBlaze * amountTitan) * (100 - _slippage)
    ) / 100;
```

If `titanPriceInBlaze` has any decimals, `adjustedBlazeAmount` will be scaled incorrectly, as extra 0's will be added.

This will make the following `UNISWAP_V2_ROUTER` call impossible, as `adjustedBlazeAmount` acts as slippage, since it's overscaled and has extra 0's, the function will always revert on Uniswap's side.

```
function swapExactTokensForTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
        amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
        require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router:
            INSUFFICIENT_OUTPUT_AMOUNT');
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, UniswapV2Library.pairFor(factory, path[0], path[1])
                , amounts[0]
        );
        _swap(amounts, path, to);
    }
```

Recommendation: Scale `titanPriceInBlaze` back to 0 decimals. Mirror the logic in `getEthPrice`.

Resolution: Fixed

5.3 Medium risk

5.3.1 BlazeBuyAndBurn.sol#receive() - _lastBurnCalledTimestamp can be set prior to currentDayInContract > 3

Severity: *Medium risk*

Context: BlazeBuyAndBurn.sol#L93

Description:

The contract implements a "lock", `swapWETHForBlazeAndBurn` can only be called after 3 days have passed since the deployment of the contract.

`EthBurnable` is the value used to calculate how much eligible ETH (in USD value) can be burned. It's calculated based of `_checkCoolDown`

```
function _checkCoolDown() private returns (uint256) {
        uint256 difference = block.timestamp - _lastBurnCalledTimestamp;
        require(difference > SWAP_FREQUENCY, "BlazeBuyAndBurn:cooldown not finished"
            );

        uint256 hoursPassed = difference / SWAP_FREQUENCY;
        uint256 EthBurnable = hoursPassed * PER_SWAP_CAP_HOURLY;
        _lastBurnCalledTimestamp = uint32(block.timestamp);
        return EthBurnable;
    }
```

`difference` is calculated based of `block.timestamp - _lastBurnCalledTimestamp`.

`_lastBurnCalledTimestamp` is set in the constructor to `_deploymentTimeStamp + 3 days;`, so 3 days after the deployment timestamp.

The issue is this isn't always the case, as in `receive`, if `buyAndBurnBalanceDollar < PER_SWAP_CAP_HOURLY` , then `_lastBurnCalledTimestamp` is set to `block.timestamp`, which isn't necessary 3 days after the deployment of the contract, meaning that technically anyone can decrease it's value since it's set in the future in the constructor.

As long as `_liquidityAdded = true` and `currentDayInContract <= 3` it's a problem.

This will affect `_checkCoolDown` as if `_lastBurnCalledTimestamp` is set in the 2nd day for example, then the `EthBurnable` will be quite a lot more than originally intended.

Recommendation: Before setting `_lastBurnCalledTimestamp` add a check for `if(currentDayInContract > 3)`

Resolution: Fixed

5.3.2 MEV opportunities in staking contract may lead to unfair reward distributions

Severity: *Medium risk*

Context: blazeStaking.sol#L157

Description:

Users stake their blaze tokens in `BlazeStaking` contract for a period between 88 and 2888 days. When they have active stake, they are eligible for rewards, that are being distributed on cycles (8 , 88 and 288 days). Step-wise jumps are red flags in a system because they mark an inconsistency in the system. A user that has staked for period of 288 days would receive the same amount for cycle 288 as other user, who has staked right before the distribution for 88 days. User can always claim rewards that have been accrued for 288 days by staking for the min period (88 days) right before distribution. This would be unfair for long-term stakers, because “expoiters” will inflate their rewards.

Recommendation: The fix may be complex. If you split `totalShares` to `totalShareStakers288` and `totalShares` you may increment `totalShareStakers288` only if the user has staked for >= 288 days and when distributing rewards for this cycle, make only those stakers eligible

Resolution: Acknowledged

5.3.3 Hardcoding `INITIAL_LP_BLAZE` amount when minting a position opens up arbitrage oppertinities

Severity: *Medium risk*

Context: BlazeBuyAndBurn.sol#476

Description:

Protocol plan to deploy titan/blaze uniswap pool and provide initial liquidity inside it. `createInitialLiquidity` function calculate corresponding amounts for blaze and titan tokens to be deposited as liquidity. There could be a problem, if contract still haven't received enough eth(weth), which is being swapped for titan and used for the openind the corresponding position:

```
    function _getTokenAmountDetails() private returns (uint256 amount0, uint256
        amount1) {
        uint256 WETHBalance = IERC20(WETH9).balanceOf(address(this));
        uint256 usableWETH = INITIAL_LP_WETH_DOLLAR_WORTH /*30_000 * 1e18*/ /
            getEthPrice();
        if (_currentDayInContract > 3) {
            if (WETHBalance < usableWETH) {
                usableWETH = WETHBalance;
            }
        } else {
            require(WETHBalance >= usableWETH, "BlazeBuyAndBurn:wait for 3 days to
                complete");
        }
        uint256 titanAmount = _swapWETHForTitan(usableWETH);

        return (titanAmount, INITIAL_LP_BLAZE); // @audit hardcoded INITIAL_LP_BLAZE
             for different amount of titanAmount opens up arbitrage oppertinities
    }
```

Imagine the following scenario:

1. If on day 3 we have only 1 WETH in the contract and someone calls `createInitialLiquidity`, `titanAmount` for the position would be worth only 1 weth (let's say $3300)
2. No matter if `titanAmount = 1 weth`, or `titanAmount = 9 weth (Around $30K, which is the cap for the position)`, we always mint `14_000 ether` blaze tokens for the position. We will use `weth` representation for amounts, instead of titan for simplicity of the calculations
3. We mint position with `1 weth` and `14_000 blaze` tokens. This means that `1 blace token = 0.00071428571` (~$2.50), which may be way larger than the prices in the auction.
4. Or if we have the cap `(9 WETH)= 14_000 blaze` => `1 blaze = 0.64285714285` ($2121)

Both result in arbitrageurs taking advantage after the position miniting. Impact is larger, because an expoiter can benefit from calling `createInitialLiquidity`, before enough liquidity has been deposited

Recommendation: Multiple solutions to prevent such problem: - Hardcode `usableWETH` to match the ration between eth and blaze that you want to start with (You can integrate it with the auction and dynamically obtain blaze prize from last mint) - Implement some balancing function, which accepts ratio and calculates corresponding blaze amount, based on current eth balance - - NOTE: For second solution you should make `createInitialLiquidity` ownable

Resolution: Acknowledged

5.3.4 If front-run `BlazeBuyAndBurn::createLiquidityPool` by creating uni pool, contract is DoS-ed

Severity: *Medium risk*

Context: BlazeAndBurn.sol#L195-L199

Description:

`BlazeBuyAndBurn` is integrated with UniswapV2 and has to use an existing pool with liquidity to execute blaze/titan swaps. In the current version of the code, the only way to set the pool is by creating one:

```
    function createLiquidityPool() external dailyUpdate {
        require(_blazeTitanPool == address(0), "BlazeBuyAndBurn:pool already exist")
            ;
        (address token0, address token1) = _getTokensDetails();
        _blazeTitanPool = IUniswapV2Factory(UNISWAP_V2_FACTORY).createPair(token0,
            token1);
    }
```

But `IUniswapV2Factory(UNISWAP_V2_FACTORY).createPair(token0, token1)` function will revert, if there is already existing pool for the pair. Result is that the contract is useless, because without `_blazeTitanPool` being set, it's main functionality is DoS-ed. Impact is that protocol looses funds, as the deployment for such large contract is not cheap. Additionally if some eth has already been sent to the contract, becomes stucked(lost).

Recommendation: Modify `createLiquidityPool` function to check if there is already existing pool:

```
    function createLiquidityPool() external dailyUpdate {
        require(_blazeTitanPool == address(0), "BlazeBuyAndBurn:pool already exist")
            ;
        (address token0, address token1) = _getTokensDetails();
+       (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (
    tokenB, tokenA);
+        address existingPool = IUniswapV2Factory(UNISWAP_V2_FACTORY).getPair[token0
    ][token1];
+        if (existingPool != address(0)){
+            _blazeTitanPool = existingPool;
+            return;
+        }
        _blazeTitanPool = IUniswapV2Factory(UNISWAP_V2_FACTORY).createPair(token0,
            token1);
    }
```

Resolution: Fixed

5.4 Low risk

5.4.1 BlazeBuyAndBurn.sol#receive() - Doesn't account for the current ETH donated when checking PER_SWAP_CAP_HOURLY

Severity: *Low risk*

Context: BlazeBuyAndBurn.sol#L85-L98

Description: When sending ETH to the contract, the `receive` has a cap check.

```
receive() external payable {
        if (msg.sender != WETH9) {
            uint256 bornFirePercentage = 0;
            if (_liquidityAdded) {
                bornFirePercentage = (msg.value * BORN_FIRE_PERCENTAGE) /
                    PERCENT_BASE;
                TransferHelper.safeTransferETH(payable(_boneFireAddress),
                    bornFirePercentage);
                uint256 buyAndBurnBalance = IERC20(WETH9).balanceOf(address(this));
                uint256 buyAndBurnBalanceDollar = (buyAndBurnBalance * getEthPrice()
                    );
                if(buyAndBurnBalanceDollar < PER_SWAP_CAP_HOURLY){
                    _lastBurnCalledTimestamp = uint32(block.timestamp);
                }
            }

            IWETH9(WETH9).deposit{value: (msg.value - bornFirePercentage)}();
        }
    }
```

If `(buyAndBurnBalanceDollar < PER_SWAP_CAP_HOURLY == true` then `_lastBurnCalledTimestamp` is set.

The issue is that `buyAndBurnBalanceDollar` is based on the WETH balance of the contract and the newly sent `msg.value` isn't taken into account, because it's wrapped on the last line of the `receive`.

This means that the real `buyAndBurnBalanceDollar` can exceed the `PER_SWAP_CAP_HOURLY`, which will affect `_checkCoolDown` as it will pass the first `require`, since `_lastBurnCalledTimestamp` isn't set in the `receive`.

```
function _checkCoolDown() private returns (uint256) {
        uint256 difference = block.timestamp - _lastBurnCalledTimestamp;
        require(difference > SWAP_FREQUENCY, "BlazeBuyAndBurn:cooldown not finished"
            );

        uint256 hoursPassed = difference / SWAP_FREQUENCY;
        uint256 EthBurnable = hoursPassed * PER_SWAP_CAP_HOURLY;
        _lastBurnCalledTimestamp = uint32(block.timestamp);
        return EthBurnable;
    }
```

Recommendation: Take into account the `msg.value - bornFirePercentage` when calculating `buyAndBurnBalanceDollar` so the value doesn't exceeed `PER_SWAP_CAP_HOURLY`.

Resolution: Acknowledged

5.4.2 Inconsistency in `DiamondHand::getCurrentDayAndCycleDetails` on how `currentDayInCycle` is calculated

Severity: *Low risk*

Context: DiamondHand.sol#L112

Description: In `DiamondHand` we have a function `getCurrentDayAndCycleDetails`, which is calculating current day, current 888 days cycle and current day of that corresponding cycle:

```
    function getCurrentDayAndCycleDetails()
        public
        view
        returns (uint256 currentDay, uint256 currentCycle, uint256 currentDayInCycle
            )
    {
        currentDay = ((block.timestamp - i_initialTimestamp) / 1 days) + 1;
        currentCycle = (currentDay / 888) + 1;
        currentDayInCycle = currentDay % 888;
    }
```

We may notice that protocol start counting the days from 1 (not zero) That means that for first cycle we will have `currentDayInCycle` in range [1; 887], but each consecutive cycle will have them in range [0; 887]. This is the case, because on day 888 after deployment, we will be steping into second cycle, but `currentDayInCycle = 888 % 888 = 0`. In current scope there is no larger impact, than simple inconsistency between first and other cycles. But if there is an integration, which is using `currentDayInCycle` for division, there could arise larger problem.

Resolution: Acknowledged

5.4.3 BlazeBuyAndBurn.sol#onlyEOA() - The invariant may not hold in the future

Severity: *Low risk*

Context: BlazeBuyAndBurn.sol#L59

Description: `onlyEOA` is a modifier that enforces `msg.sender == tx.origin`. This is done so that only EOA's can call functions with the modifier attached to them.

```
modifier onlyEOA() {
        require(msg.sender == tx.origin);
        _;
    }
```

Currently, this holds, however EIP7702 (which extends [EIP-3074])(https://eips.ethereum.org/EIPS/eip-3074) introduces a way for contracts to act as EOA's.

> Allowing `tx.origin` to set code enables simple transaction batching, where the sender of the outer transaction would be the signing account. The ERC-20 approve-then-transfer pattern, which currently requires two separate transactions, could be completed in a single transaction with this proposal.
>
> Once code exists in the EOA, it's possible for self-sponsored EIP-7702 transactions to have `msg.sender == tx.origin` anytime the code in the EOA dispatches a call.

EIP-3074 specifically states: > Therefore this EIP breaks that invariant and so it affects smart contracts containing require(msg.sender == tx.origin) checks. This check is used for at least three purposes:

You can read further about EIP-7702 here and also here

You can also find the same issue in a Sherlock contest

Recommendation: Remove the modifier

Resolution: Acknowledged

5.4.4 BlazeBuyAndBurn.sol#_checkCoolDown() - Returns `EthBurnable` based on 15 minutes not 1 hour

Severity: *Low risk*

Context: buyAndBurnConstants.sol#L30

Description: The function is supposed to return `EthBurnable` based on `hoursPassed`.

If we look at the function, we see that `hoursPassed` is calculated based of `difference / SWAP_FREQUENCY`.

```
uint256 hoursPassed = difference / SWAP_FREQUENCY;
```

The issue is that `SWAP_FREQUENCY = 15 minutes`, not 1 hour, so `hoursPassed` will return how many 15 minute intervals have passed, not how many 1 hour intervals have passed. This will increase `EthBurnable` 4x, heavily inflating the value.

Recommendation: Change `hoursPassed` to `minutesPassed`, if you want to have 15 min, or change `SWAP_FREQUENCY` to 1 hour.

Resolution: Fixed

5.4.5 OZ's Context `_msgSender` is mixed with `msg.sender`

Severity: *Low risk*

Context: blazeStaking.sol#L130

Description: 5 times using `msg.sender` instead of `_msgSender` in `blazeStaking.sol` The same inconcistency is also present in the other contracts.

Recommendation: Use `_msgSender` on all places.

Resolution: Fixed

5.4.6 `nextActiveEventIndex` may become outdated

Severity: *Low risk*

Context: BlazeBonfire.sol#L118-L121

Description: If the whole `INTERVAL_BETWEEN_EVENTS` has passed and funds haven't been utilized, `nextActiveEventIndex` won't be incremented, but new cycle for new `currentDate` will start. We couldn't find impact from this, but `nextActiveEventIndex` would be `currentDate - 1` from so on. Also `events[{date for which has not been burned}].active` would be always `true`.

Recommendation: You may consider checking if `nextActiveEventIndex = currentDate - 1` inside `initiateBonfireBurn` and if so - incrementing it and marking previous event entry as inactive

Resolution: Fixed

5.4.7 `BlazeBuyAndBurn.sol#swapWETHForBlazeAndBurn()` - Doesn't follow CEI pattern

Severity: *Low risk*

Context: BlazeBuyAndBurn.sol#L161

Description: On the last lines of the function, it first transfers `incentive` to `msg.sender` and then increases `_totalEthUsed`, which doesn't follow the CEI pattern.

Currently, this has no effect, as `_totalEthUsed` is only used in `getTotalEthUseForBurn` which is a view function and isn't called anywhere else in the codebase, but if in the future the function is used for some calculation or a third party protocol uses the function for it's calculation `msg.sender` can do a read-only reentrancy to read the yet not updated value of `_totalEthUsed`.

Recommendation: Increase `_totalEthUsed` first then transfer ETH to `msg.sender`.

```
        _totalEthUsed += ethAmount;
TransferHelper.safeTransferETH(payable(msg.sender), incentive);
```

Resolution: Fixed

5.4.8 BlazeBuyAndBurn.sol - `dailyUpdate` modifier is redundant in several functions

Severity: *Low risk*

Context: BlazeBuyAndBurn.sol#L404BlazeBuyAndBurn.sol#L423

Description: The `_dailyUpdate` modifier is used to calculate and set, if required, the `_currentDayInContract` .

The modifier is also attached to `_swapWETHForTitan` and `_swapTitanToBlaze`, which are both internal functions used only in functions that have `dailyUpdate` attached to them already.

This means that the calculation for the `_currentDayInContract` will be calculated twice, the second time being unnecessary, as the value has already been calculated in the first call, while the second one just increases the gas cost of the tx.

Recommendation: Remove `dailyUpdate` from `_swapWETHForTitan` and `_swapTitanToBlaze`

Resolution: Fixed

5.4.9 `DiamonHand.sol#getMintedNFT()` - The function can be gamed by two addresses

Severity: *Low risk*

Context: DiamonHand.sol#L94-L96

Description: `getMintedNFT` gets the min value between the current balance of `_user` and his `userMintedNFT`.

```
function getMintedNFT(address _user) external view returns(uint256 _nftCount){
        uint holdingCount = IDiamondNFT(_nftContractAddress).balanceOf(_user);
        uint mintingCount = IDiamondNFT(_nftContractAddress).userMintedNFT(_user);

        return Math.min(holdingCount, mintingCount);
    }
```

The function in DiamonHand retrieves the value and then divides in by 28.

```
function getMintedNFT(address user) public view returns (uint256) {
        return dNFT.getMintedNFT(user) / 28;
    }
```

The function inside `DiamonHandWrapper` can very easily be gamed by several addresses that have a minimum of 28 minted NFT's.

Example: 1. Alice and Bob both have minted 28 NFT's and their current balance is both 28. 2. Alice sells all her NFT's, so her `balanceOf = 0`, but her `userMintedNFT = 28`. 3. Bob calls `participate` and has 1 ticket. His `balanceOf = 28`. 4. After Bob participates he sends all 28 of his NFT's to Alice. 5. Alice now has `balanceOf = 28` and `userMintedNFT = 28`, she calls `participate` and gets 1 ticket.

The attack is infinitely repetable.

Recommendation: Force users to transfer their NFT's when they participate.

Resolution: Acknowledged

5.4.10 `BlazeBuyAndBurn.sol#_swapWETHForTitan()` - The swap has effectively no deadline

Severity: *Low risk*

Context: BlazeBuyAndBurn.sol#L414

Description: `_swapWETHForTitan` makes a call to the `UNISWAP_V3_ROUTER` to swap WETH for Titan.

The issue is that when the `ExactInputParams` are built, `deadline` is set to `block.timestamp + 1`. This effectively means that the swap has no deadline, because when the tx is executed it will use `block.timestamp + 1` as the deadline and since the `block.timestamp` is the timestamp of the current block, this effectively means no deadline, whenever the tx is executed, the deadline will be that timestamp + 1.

Recommendation: Let the caller of the function specify a deadline.

Resolution: Acknowledged

5.5 Informational

5.5.1 `EIGHTH_DAY_SHARE_RATE_DECREASE_PERCENTAGE` in docs is set to `1.44`

Severity: *Informational*

Context: constants.sol#L21

Description: `EIGHTH_DAY_SHARE_RATE_DECREASE_PERCENTAGE` in docs is set to `1.44%`, but in code is `1.26%`

Resolution: Fixed

5.5.2 Use error messages when using `require`

Severity: *Informational*

Context: BlazeBuyAndBurn.sol#L59

Description: Use error messages when using ‘require, so you can be more descriptive.

Resolution: Fixed

5.5.3 `distributeFeeRewardsForAll` will always attempt to transfer 0 ETH if `lastCycleDistributionPortion = 0`, which happens when `_totalUndistributedCollectedFees = 0`. This is unnecessary as it will just consume more gas.

Severity: *Informational*

Context: blazeStaking.sol#L197

Description: `distributeFeeRewardsForAll` will always attempt to transfer 0 ETH if `lastCycleDistributionPortion = 0`, which happens when `_totalUndistributedCollectedFees = 0`. This is unnecessary as it will just consume more gas.

Resolution: Fixed

5.5.4 `DAILY_SWAP_CAP` is unreachable because of division and then multiplication. The division will round down the value and then the rounded value will be added to `_dayToUsedEth`

Severity: *Informational*

Context: BlazeBuyAndBurn.sol#L174

Description: `DAILY_SWAP_CAP` is unreachable because of division and then multiplication. The division will round down the value and then the rounded value will be added to `_dayToUsedEth`

```
 uint256 ethAmount = (remainingEthUSDValue) / getEthPrice();

...

_dayToUsedEth[_currentDayInContract] += ethAmount * getEthPrice();
```

Resolution: Acknowledged

5.5.5 Typos in `BlazeBuyAndBurn.sol#receive()`

Severity: *Informational*

Context: BlazeBuyAndBurn.sol#L84-L98

Description: Typos in `BlazeBuyAndBurn.sol#receive()`

```
receive() external payable {
        if (msg.sender != WETH9) {
            uint256 bornFirePercentage = 0;
            if (_liquidityAdded) {
                bornFirePercentage = (msg.value * BORN_FIRE_PERCENTAGE) /
                    PERCENT_BASE;
                TransferHelper.safeTransferETH(payable(_boneFireAddress),
                    bornFirePercentage);
                uint256 buyAndBurnBalance = IERC20(WETH9).balanceOf(address(this));
                uint256 buyAndBurnBalanceDollar = (buyAndBurnBalance * getEthPrice()
                    );
                if(buyAndBurnBalanceDollar < PER_SWAP_CAP_HOURLY){
                    _lastBurnCalledTimestamp = uint32(block.timestamp);
                }
            }

            IWETH9(WETH9).deposit{value: (msg.value - bornFirePercentage)}();
        }
    }
```

Should be bonfire not bornfire.

Resolution: Fixed

5.5.6 CEI isn't followed in `BlazeBuyAndBurn.sol#receive()`

Severity: *Informational*

Context: BlazeBuyAndBurn.sol#L84-L98

Description: CEI isn't followed in `BlazeBuyAndBurn.sol#receive()`. Move `TransferHelper.safeTransferETH(payable(_boneFireAddress), bornFirePercentage);` to the bottom of the function.

Resolution: Acknowledged

5.5.7 `_userClaimSharesIndex` always starts at 0, which just wastes gas as all counts in the protocol start from 1.

Severity: *Informational*

Context: blazeStaking.sol#L547

Description: `_userClaimSharesIndex` always starts at 0, which just wastes gas as all counts in the protocol start from 1.

Resolution: Acknowledged

5.5.8 `TransferHelper.safeApprove(TITANX_TOKEN, UNISWAP_V2_ROUTER, amountTitan);` on each `_swapTitanToBlaze` is redundant and will only waste gas, because contract has already approved `UNISWAP_V2_ROUTER` to use `type(uint256).max` when adding initial liquidity.

Severity: *Informational*

Context: BlazeBuyAndBurn.sol#L424

Description: `TransferHelper.safeApprove(TITANX_TOKEN, UNISWAP_V2_ROUTER, amountTitan)` ; on each `_swapTitanToBlaze` is redundant and will only waste gas, because contract has already approved `UNISWAP_V2_ROUTER` to use `type(uint256).max` when adding initial liquidity.

Resolution: Acknowledged

5.5.9 Cache the value of `getEthPrice` inside `swapWETHForBlazeAndBurn` to safe gas, instead of calucating it two times.

Severity: *Informational*

Context: BlazeBuyAndBurn.sol#L174

Description: Cache the value of `getEthPrice` inside `swapWETHForBlazeAndBurn` to safe gas, instead of calucating it two times.

Resolution: Fixed