EGIS SECURITY

# SECURITY REVIEW FOR INFERNO VERSION 1.0

# Table of Contents

**1 About Egis Security** **3**

**2 Disclaimer** **3**

**3 Risk classification** **3**
3.1 Impact . . . 3
3.2 Likelihood . . . 3
3.3 Actions required by severity level . . . 3

**4 Executive summary** **4**

**5 Findings** **5**
5.1 Medium risk . . . 5
5.1.1 `_swapBlazeForInferno` has effectively no slippage . . . 5
5.1.2 Anyone can call `distributeTitanXForBurning` during a cycle execution and mess up expected allocations . . . 6
5.1.3 `InfernoBuyAndBurn` daily allocation doesn't account for missed days . . . 7
5.1.4 Unsafe setting of `intervals[_lastInterval].amountAllocated`, which may be above contract titan balance . . . 7
5.2 Low risk . . . 9
5.2.1 `getBlazeQuoteForTitanX` doesn't check `token0 == titanX` . . . 9
5.2.2 `InfernoBuyAndBurn#Interval::amountBurned` is unclear . . . 9
5.2.3 Burned inferno fees aren't tracked . . . 10
5.2.4 Consider providing `sqrtPriceX96` from outside, because uniswapV2 pool reserves can be manipulated at time of deployment . . . 10
5.3 Informational . . . 11
5.3.1 Typo in `Inferno.sol` construnctor natspec . . . 11
5.3.2 Consider removing `indexed` keyword from amount fields in the events. The probability of someone searching by specific token amounts is very low . . . 11
5.3.3 Consider emitting `InfernoMinting#MintExecuted` inside mint function with `adjustedAmount`, or specifying the name of the event argument. . . 11
5.3.4 `InfernoBuyAndBurn#_intervalUpdate` consider treating `timeElapseSinceLastBurn == INTERVAL_TIME` as successfully passed period. Currently if exactly 28 minutes . . . 11
5.3.5 By using dust amount for `InfernoMinting#mint` user can skip burning titanX tokens . . . 12
5.3.6 Division before multiplication in `_calculateIntervals` . . . 12

# 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 >$70,000 in winnings and multiple solo/team audits.

# 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 | Inferno |
| Repository | https://github.com/Kuker-Labs/Inferno-Contracts |
| Commit hash | 0668f71e60505824440d87f521466c66891f5969 |
| Resolution | 1c818b6f741292091ea31dbfaea2aa2c6f305458 |
| Documentation | https://docs.inferno.win/inferno |
| Methods | Manual review |

**Scope**

| |
|---|
| /src/Inferno.sol |
| /src/InfernoBuyAndBurn.sol |
| src/InfernoMinting.sol |
| src/library/OracleLibrary.sol |
| src/const/BuyAndBurnConst.sol |

**Issues Found**

| | |
|---|---|
| Critical risk | 0 |
| High risk | 0 |
| Medium risk | 4 |
| Low risk | 4 |
| Informational | 6 |

# 5 Findings

## 5.1 Medium risk

### 5.1.1 `_swapBlazeForInferno` has effectively no slippage

**Severity:** *Medium risk*

**Context:** InfernoBuyAndBurn.sol#L353

**Description:** If we take a look into `_swapBlazeForInferno` function, which is responsible for swapping `blaze` for `inferno` tokens, we can notice that we calculate `adjustedInfernoAmount` to be used as `minAmountOut`:

```
uint256 adjustedInfernoAmount = (expectedInfernoAmount * (100 - 97)) / 100;
```

The value is calculated as 3% of the original `expectedInfernoAmount`. This means that we allow for 97% slippage from the `expectedInfernoAmount`, which is a large amount and can suffer sanwich attacks and bricking protocol tokenomics. It is hard to estimate the impact and cost of a sandwich attack execution, because it depends on the `titanX` inflow and the future liquidity of `blaze`/`inferno` uniswapV3 pool. That's why we leave it as Medium severity

**Recommendation:** Calculate `adjustedInfernoAmount` as:

```
   uint256 adjustedInfernoAmount = (expectedInfernoAmount * 97) / 100;
```

**Resolution:** Fixed.

### 5.1.2 Anyone can call `distributeTitanXForBurning` during a cycle execution and mess up expected allocations

**Severity:** *Medium risk*

**Context:** InfernoBuyAndBurn.sol#L227-L231

**Description:** If we take a look at protocol expected calculations we see that for the value of $540,000.00 deposited titan tokens at the end of the mint window (Saturday at 2pm UTC), team expect to allocate corresponding percentages based on the original value. For days Sunday - Wednesday - $21,600.00 each day, which is 4% of $540,000.00 (totalTitanXDistributed). The problem is that when we burn those $21,600.00 on Sunday, anyone can call distributeTitanXForBurning, which will update totalTitanXDistributed to value of $518,400. So on Monday instead of burning $21,600.00, we will have 4% from $518,400, which is = 20,736. The value will drastically drop when we reach Friday.

**Recommendation:** Implement an access control to `distributeTitanXForBurning` so only `InfernoMinting` can call it (which is a time window of 24 hours every week)

**Resolution:** Fixed.

### 5.1.3 `InfernoBuyAndBurn` daily allocation doesn't account for missed days

**Severity:** *Medium risk*

**Context:** InfernoBuyAndBurn.sol#L375-L379

**Description:** Protocol has a mechanism to buy and burn inferno tokens with titanX tokens, which have been deposited. Documented tokenomics are such that we have a corresponding daily burn allocation. This meas that each week day has different %. This % is used to calculate what amount of the `titanX` tokens should be used for the buy and burn. Here are the different allocation percentages for the different week days:

```
    function getDailyTitanXAllocation() public view returns (uint32
        dailyBPSAllocation) {
        dailyBPSAllocation = 400; // 4 %

        uint8 weekDay = currWeekDay();

        if (weekDay == 4 || weekDay == 5) {
            dailyBPSAllocation = 1500; // 15%
        } else if (weekDay == 3) {
            dailyBPSAllocation = 1000; // 10%
        }
    }
```

The problem is that if we have missed two days with 4% `dailyBPSAllocation` and call `_calculateIntervals` on a day with 15% `dailyBPSAllocation`, we will have a total of 45% (3 * 15), instead of 23% (15 + 4 + 4). The following breaks predefined protocol tokenomics.

**Recommendation:** Consider implementing a mechanism, which calculate missed days from last `lastBurnedIntervalStartTimestamp` and use the planned `dailyBPSAllocation` for those days.

**Resolution:** Fixed.

### 5.1.4 Unsafe setting of `intervals[_lastInterval].amountAllocated`, which may be above contract titan balance

**Severity:** *Medium risk*

**Context:** InfernoBuyAndBurn.sol#L383-L385

**Description:** **NOTE** The following is a combination of two roots: - hardcoded start time in deploy scripts - unsafe setting of `intervals[_lastInterval].amountAllocated`, which may be above contract titan balance

`infernoMintingStartTimestamp` is used inside `InfernoBuyAndBurn` to calculate intervals passed since the "strat time", or the last time a burn has been triggered. The problem is that we have hardcoded the variable to a value is the past 17.07.2024. If we assume that protocol will be deployed at the earliest on 22.07.2024, we have ~ 5 days in the past, which are 257 intervals. Another think is that the protocol may not reach the required `INITIAL_TITAN_X_FOR_LIQ` for another 5 days. Now we would have 10 days, before `intervalUpdate` inside `swapTitanXForInfernoAndBurn` is called. Now imagine we call `swapTitanXForInfernoAndBurn` for first time after 10 days and `weekDay = 4` (daily allocation = 15%) The following will result in `intervals[_lastInterval] = 150% of totalTitanXDistributed`

The following will result in reverts when `swapTitanXForInfernoAndBurn` is called, because protocol will try swapping more titanx than it actually have. Impact of the DoS depends of the expected titanX inflow of funds per interval.

**Recommendation:** - Instead of hardcoding timestamp, pass a value, which would be the added to `block.timestamp` on deployment

- When you calculate `_calculateIntervals#_totalAmountForInterval` check if contract have enough balance:

```
    function _calculateIntervals(uint256 timeElapsedSince)
        internal
        view
        returns (uint32 _lastIntervalNumber, uint128 _totalAmountForInterval, uint16
            missedIntervals)
    {
      ...
        uint128 additionalAmount = _amountPerInterval * missedIntervals;
        _totalAmountForInterval = _amountPerInterval + additionalAmount; // all
            missed intervals amounts
+        if (_totalAmountForInterval > totalTitanXDistributed){
+          _totalAmountForInterval = totalTitanXDistributed;
+        }
    }
```

**Resolution:** Fixed.

## 5.2 Low risk

### 5.2.1 `getBlazeQuoteForTitanX` doesn't check `token0 == titanX`

**Severity:** *Low risk*

**Context:** InfernoBuyAndBurn.sol#L304

**Description:** Currently the function doesn't check if token0 == titanX and just assumes that it is. This is currently correct on both Sepolia and Mainnet, but for the sake of consistency and if the protocol decides to deploy on other chains in the future, a similar check should be implement, like the one in `Inferno#_createBlazeInfernoPool`.

```
        (uint112 res0, uint112 res1,) = IUniswapV2Pair(_blazeTitanXPool).getReserves
            ();
        address token0V2 = IUniswapV2Pair(_blazeTitanXPool).token0();

        (uint112 resIn, uint112 resOut) = token0V2 == _titanX ? (res0, res1) : (res1
            , res0);

        uint256 blazeAmount = IUniswapV2Router02(UNISWAP_V2_ROUTER).getAmountOut(
            INITIAL_TITAN_X_FOR_LIQ, resIn, resOut);
```

**Recommendation:** - Add a similar check like the one in `Inferno#_createBlazeInfernoPool`. Or - Remove `getBlazeQuoteForTitanX` as it is not used anywhere

**Resolution:** Acknowledged.

### 5.2.2 `InfernoBuyAndBurn#Interval::amountBurned` is unclear

**Severity:** *Low risk*

**Context:** InfernoBuyAndBurn.sol#L153

**Description:** There is a struct `Interval` inside `InfernoBuyAndBurn`, which holds: - amountAllocated - amountBurned

`amountAllocated` represents titan token amount for the given interval, which is used to be swapped for `blaze`, which then swapped for `inferno`. **NOTE** that we also subtract an `incentive` amount from `amountAllocated`, which is transferred to the caller of the function:

```
Interval storage currInterval = intervals[lastIntervalNumber];

        currInterval.amountBurned = currInterval.amountAllocated;

        uint256 incentive = (currInterval.amountAllocated * INCENTIVE_FEE) /
            BPS_DENOM; // 1.5% incentive fee
```

We notice that `amountBurned` is set to `amountAllocated`. But we don't burn the whole `amountAllocated`, because 1.5% are being transferred to the caller.

Also `amountBurned` is assigned to titanX value, but we swap titaX and burn Inferno. Also `BuyAndBurn` event is being emitted with `infernoAmount`, which is the inferno obtained from the swap for `infernoBurnt`. But we burn the whole inferno balance of the token, which may be larger than `infernoAmount`

**Recommendation:** Consider assigning `titanXToSwapAndBurn` to `currInterval.amountBurned` Or if you want to track `inferno` burned, assign `uint256 infernoToBurn = infernoToken.balanceOf(address(this));` to `currInterval.amountBurned`.

For the event -> make `burnInferno` function return `uint256 burnedAmount` and use the value for the second arg in `BuyAndBurn` event.

**Resolution:** Acknowledged.

### 5.2.3 Burned inferno fees aren't tracked

**Severity:** *Low risk*

**Context:** InfernoBuyAndBurn.sol#L239-L255

**Description:** `burnFees` collects all the fees from the UniV3 position and then burns them. When the inferno tokens are burned they aren't tracked in `totalInfernoBurnt`.

The comment above the state variable states: > /// @notice Total amount of Inferno tokens burnt

Currently this isn't true, as the variable will only track inferno that is burned through `burnInferno`.

**Recommendation:** Track inferno fees that are burned.

```
totalInfernoBurnt = totalInfernoBurnt + infernoAmount;
```

**Resolution:** Fixed

### 5.2.4 Consider providing `sqrtPriceX96` from outside, because uniswapV2 pool reserves can be manipulated at time of deployment

**Severity:** *Low risk*

**Context:** Inferno.sol#L128-L143

**Description:** When an Inferno contract is created, it creates a blaze/inferno uniswapV3 pool. To determine the price between Blaze and inferno, the team is querying current reserves (price) between Blaze and titanx from their pool in uniswapV2. The problem is that `getAmountOut` will take the current reserves ratio, which can easily be manipulated with a sandwich attack, which will mess up the ratio expected from the team.

There is no big impact from the following, because when later liquidity is provided inside the `InfernoBuyAndBurn` contract, the tick price will be moved accordingly. However, there is an edge case where an exploiter can set a large cardinality for the blaze/inferno pool, which will hold the fake (manipulated price) for a long time. When the owner of `InfernoBuyAndBurn` provides liquidity to the pool, the tick will be moved, but the twap price returned from uniswap oracle will be influenced by the old `sqrtPrc`, which may result in receiving less inferno tokens when first `swapTitanXForInfernoAndBurn` is called, because `getInfernoQuoteForBlaze` is using the orcale twap.

**Recommendation:** Provide sqrtPriceX96 as an argument to be sure there is no big difference.

**Resolution:** Acknowledged.

## 5.3 Informational

### 5.3.1 Typo in `Inferno.sol` construnctor natspec

**Severity:** *Info risk*

**Context:** Inferno.sol#L55

**Description:** Typo in `Inferno.sol` construnctor natspec: `poll` should be `pool`

```
     * @param _blazeTitanXPool The BLAZE/TITANX UniswapV2 poll
```

**Resolution:** Fixed.

### 5.3.2 Consider removing `indexed` keyword from amount fields in the events. The probability of someone searching by specific token amounts is very low

**Severity:** *Info risk*

**Context:** InfernoMinting.sol#L69-L72

**Description:** Most probably noone will search on specific amount minted, but each time event is emitted, gas paid is higher than argument with no **`indexed`** key word. `poll` should be `pool`

**Resolution:** Fixed.

### 5.3.3 Consider emitting `InfernoMinting#MintExecuted` inside mint function with `adjustedAmount`, or specifying the name of the event argument.

**Severity:** *Info risk*

**Context:** InfernoMinting.sol#L124

**Description:** Currently the argument name is `amount`. But this corresponds to titnanX amount deposited by the user, which is being distributed between `dead` address, `GENESIS` address and `InfernoBuyAndBurn` address. Consider modifing the name of the arg to clarify what is it's purpose, because someone may assume that `MintExecuted::amount` is the amount of the minted token, which is something else because it is being further adjusted.

**Resolution:** Acknowledged.

### 5.3.4 `InfernoBuyAndBurn#_intervalUpdate` consider treating `timeElapseSinceLastBurn == INTERVAL_TIME` as successfully passed period. Currently if exactly 28 minutes

**Severity:** *Info risk*

**Context:** InfernoBuyAndBurn.sol#L420

**Description:** `InfernoBuyAndBurn#_intervalUpdate` consider treating `timeElapseSinceLastBurn == INTERVAL_TIME` as successfully passed period. Currently if exactly 28 minutes (1 period) have been passed from the previous burn, we will revert the tx. **NOTE** If you change it, you should also change comparison operator inside `_calculateMissedIntervals`

**Resolution:** Acknowledged.

### 5.3.5 By using dust amount for `InfernoMinting#mint` user can skip burning titanX tokens

**Severity:** *Info risk*

**Context:** InfernoMinting.sol#L153-L159

**Description:**

```
        unchecked {
             uint256 titanXForGenesis = (_amount * GENESIS_BPS) / BPS_DENOM;
             uint256 titanXToBurn = (_amount * TITAN_X_BURN_BPS) / BPS_DENOM;

             newAmount = _amount - titanXForGenesis - titanXToBurn;

             titanX.safeTransferFrom(msg.sender, DEAD_ADDR, titanXToBurn);
             titanX.safeTransferFrom(msg.sender, GENESIS_WALLET, titanXForGenesis);
         }
```

If `_amount * GENESIS_BPS` provided by user is `< BPS_DENOM` the calculation will be rounded down to 0 and user will skip sending corresponding value to genesis wallet, or burning it.

**Resolution:** Fixed.

### 5.3.6 Division before multiplication in `_calculateIntervals`

**Severity:** *Info risk*

**Context:** InfernoBuyAndBurn.sol#L381-L383

**Description:** Division before multiplication in `_calculateIntervals` may lead to precision loss or rounding issues. If `dailyAllcation` is a small value, we might multiply it first by `missed Intervals` and then divide the result with `INTERVALS PER DAY`

**Recommendation** Check if `_amount` is > BPS_DENOM / GENESIS_BPS

**Resolution:** Acknowledged