marchev

Inferno
Security Review

Version 1.1

Reviewed by:
Martin Marchev

23 July 2024

Table of Contents

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

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. He is currently ranked #43 on Immunefi's 90-day leaderboard. Martin has responsibly disclosed vulnerabilities in live protocols 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.

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

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 Inferno

Inferno is an ERC-20 token with buy-and-burn mechanics and custom tokenomics, designed for value management through controlled supply. Inferno is minted exclusively using TitanX. It incorporates mechanisms to support token value and aims to offer a predictable and engaging token minting experience.

2.2 Synopsis

Project Name	Inferno
Project Type	ERC-20
Repository	Inferno-Contracts
Commit Hash	9a0dd9c3...b7fd01e0
Review Period	15 July 2024 - 18 July 2024

2.3 Issues Found

Severity	Count
Critical Risk	0
High Risk	3
Medium Risk	3
Low Risk	7
Informational	6
Total Issues	**19**

3 Findings

High Risk

[H-1] Initial liquidity could be stolen via an arbitrage

Context: InfernoBuyAndBurn.sol#L205-L242

Description: The `createPoolAndAddLiquidity()` function can be called by anyone. It contains an important parameter `_amountTotalMin` (incorrectly named because it sets the min amount of Blaze the contract is willing to accept after the TitanX/Blaze swap) which plays the role of slippage protection. Once enough TitanX is accumulated in the contract a malicious actor can perform an arbitrage and steal initial liquidity from the protocol. Please note that the TitanX/Blaze UniV2 pool does not contain much liquidity (~$65K at the time of this report) and thus any medium-sized TitanX holder can execute the attack.

Example scenario:

1. Alice swaps 150B of TitanX for Blaze in the TitanX/Blaze UniV2 pool. This inflates the price of Blaze due to its lowered reserve in the pool.
2. Once the protocol accumulates 50B of TitanX, Alice calls `createPoolAndAddLiquidity(block.timestamp, 0)`. Notice the second parameter - this effectively bypasses any slippage protection.
3. The protocol swaps its TitanX tokens for Blaze tokens but it gets much fewer Blaze tokens due to its inflated price in the UniV2 pool. This inflates the Blaze token price even further.
4. Alice swaps back her Blaze tokens for TitanX tokens effectively getting the difference from the arbitrage and stealing funds from the protocol initial liquidity.

Recommendation: Restrict the `createPoolAndAddLiquidity()` function so that only the `owner` of `InfernoBuyAndBurn` can call it. Otherwise, there is no reliable way to add slippage protection for the TitanX/Blaze swap since any data based on which the slippage protection would be based is on-chain and thus manipulable.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[H-2] Large portion of the protocol bootstrap liquidity could be stolen by creating the UniswapV3 Blaze/Inferno pool with a heavily skewed price

Context: InfernoBuyAndBurn.sol#L222-L237

Description: A significant portion of the protocol's bootstrap liquidity could be stolen if a malicious actor creates the UniswapV3 Blaze/Inferno pool with a skewed price. The current implementation delays the creation of the Blaze/Inferno UniV3 pool until at least 50B TITANX is accumulated (worth ~$12K). Due to Uniswap's permissionless nature, anyone can create a pool as soon as a token is deployed. Setting the initial price (`sqrtPriceX96`) is critical, and a malicious actor can exploit this by creating the pool with an inflated Inferno price before the protocol accumulates 50B TITANX. This enables the attacker to steal the deposited Blaze liquidity once the first mint cycle ends.

Example scenario:

1. The Inferno token is deployed.
2. Alice creates the UniV3 pool with the token ratio 3700 Blaze / 5B Inferno (10x less than intended by the project).
3. Alice mints 5B Inferno during the initial mint cycle.
4. Alice claims her 5B Inferno once the initial mint cycle completes.
5. Alice swaps her 5B Inferno and extracts ~3700 Blaze from the UniV3 for 10x less than the intended price.

This issue arises due to broken liquidity protection during LP formation. The `_minus10Perc()` function returns 10% of a given amount instead of 90%.

Recommendation: Create the Blaze/Inferno UniV3 pool as early as possible with the intended Blaze/Inferno price. Ideally, this should occur atomically in the `DeployInferno.sol` script once the `Inferno` token is deployed, preventing a malicious actor from creating the UniV3 pool first.

Additionally, the function to add liquidity must be executed via Flashbots/private mempool to avoid a malicious actor disrupting it through a sandwich attack.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[H-3] Blaze/Inferno UniswapV3 pool creation can by griefed

Context: InfernoBuyAndBurn.sol#L222-L237

Description: A malicious actor can disrupt the creation of the Blaze/Inferno UniswapV3 pool by preemptively creating the pool with an excessively inflated Inferno token price. This inflated price triggers slippage protection every time the createPoolAndAddLiquidity() function is called, causing the transaction to fail. Notably, creating the pool requires no initial liquidity, making the attack cost-free for the attacker.

Example scenario:

1. The Inferno token is deployed.
2. Alice creates the UniV3 pool with a highly inflated Inferno price, e.g., 1 Inferno = 3700e36 Blaze.
3. The protocol accumulates enough TitanX to bootstrap the Blaze/Inferno UniV3 pool and someone calls `createPoolAndAddLiquidity()`.

The `POSITION_MANAGER.mint(params)` function activates slippage protection, causing the transaction to fail. Consequently, the protocol cannot initiate the buy-and-burn mechanics.

The protocol could add shallow liquidity to the pool and adjust the price through trading. However, the attacker can counter this by adding their own liquidity. If the attacker adds sufficiently deep liquidity, it can prevent the protocol from initializing the mechanism altogether.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

Medium Risk

[M-1] The TitanX burn allocations incorrectly use the contract deployment time as cutoff time

Context: InfernoBuyAndBurn.sol#L289

The current implementation incorrectly calculates the current day since the contract deployment. Instead it calculates the number of whole days since deployment, leading to an inaccurate cutoff time for daily TitanX burn allocations.

```
function currentDayInContract() public view returns (uint32 currentDay) {
    currentDay = uint32(((block.timestamp - deploymentTimeStamp) / 1 days) + 1);
}
```

This incorrect calculation results in the daily TitanX burn allocation using the deployment timestamp hour as the cutoff hour for the burn allocations instead of the intended 2pm UTC, as specified in the protocol whitepaper.

Here are some examples from a PoC to illustrate the issue:

```
Ran 1 test for test/Integration.t.sol:IntegrationTest
[PASS] test_titanx_allocation_is_incorrect() (gas: 20657)
Logs:
  Deploy @ 1721368800 (Fri Jul 19 2024 06:00:00 UTC)
  TitanX Burn Allocation @ Sun Jul 21 2024 13:30:00 UTC: expected=1500 actual=400
  TitanX Burn Allocation @ Sun Jul 21 2024 14:30:00 UTC: expected=400 actual=400
  TitanX Burn Allocation @ Thu Jul 25 2024 05:30:00 UTC: expected=400 actual=400
  TitanX Burn Allocation @ Thu Jul 25 2024 06:30:00 UTC: expected=400 actual=1000
```

This miscalculation can lead to user confusion, misaligned market strategies, and potential damage to the protocol's reputation and trust.

Recommendation: Calculate the current day correctly using only `block.timestamp`, and ensure the cutoff hour is accurately set at 2pm UTC.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[M-2] Dangerous use of deadline parameter for swaps

Context: InfernoBuyAndBurn.sol#L357, InfernoBuyAndBurn.sol#L383

Description: The protocol is using `block.timestamp`-based values as the `deadline` argument when interacting with `IUniswapV2Router02` and `ISwapRouter`. This completely defeats the purpose of using a deadline check.

Actions in the Uniswap `NonfungiblePositionManager` contract are protected by a deadline parameter to limit the execution of pending transactions. Functions that modify the liquidity of the pool check this parameter against the current block timestamp to discard expired actions.

`InfernoBuyAndBurn#_swapTitanForBlaze()` and `InfernoBuyAndBurn#_swapInfernoForBlaze()` functions provide `block.timestamp + XX` as the argument for the deadline parameter in their calls to the Uniswap routers.

Using deadlines based on `block.timestamp` as the deadline are effectively a no-operation that has no effect and provides no protection. Since `block.timestamp` will take the timestamp value when the transaction gets mined, the check will end up comparing `block.timestamp` against the same value (see here).

Recommendation: Add a deadline parameter to these functions and forward this parameter to the corresponding underlying call to the Uniswap router contracts.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[M-3] Malicious actor can undermine the burn rate on mint days

Context: InfernoBuyAndBurn.sol#L426-L428

Description: The TitanX burn allocation is calculated only once per day. A malicious actor could exploit this behavior to undermine the burn rate of the protocol.

Example scenario:

1. On Friday at 13:30 UTC (30 mins before the minting cycle starts) the accumulated TitanX in the `InfernoBuyAndBurn` contract is 100B
2. At 14:00 UTC Alice calls the `swapTitanXForInfernoAndBurn()` function. By calling it, the `_dailyUpdate()` calculates the daily allocation: `15% * 100B = 15B`
3. At 14:05 UTC Bob mints Inferno tokens for 900B TitanX

Expected behavior: the newly distributed TitanX should be added to the remaining intervals of the Friday daily burn allocation

Actual behavior: the newly distributed TitanX will be allocated for burning in the upcoming day (Saturday)

While the Saturday burn rate is also 15%, the bug in the burning mechanism means that tokens minted on Fridays may not get burned the same day, which reduces the intended scarcity. This creates a downward price pressure which may disincentivize people from minting Inferno tokens which is a key goal for the protocol. Therefore, this issue can significantly impact the token's market dynamics and should be considered of medium severity.

Recommendation: update the implementation so that any new TItanX amount that is distributed to to the `InfernoBuyAndBurn` contract is included in the remaining intervals for the day on a pro-rata basis.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

Low Risk

[L-1] Flashloan sandwich attack can undermine the buy-and-burn mechanism

Context: InfernoBuyAndBurn.sol#L191

Description: The protocol's buy-and-burn functionality, which operates on a permissionless 28-minute schedule, is susceptible to manipulation through a flashloan sandwich attack. Since anyone

can call the `swapTitanXForInfernoAndBurn()`, an attacker can influence the swap process to reduce the output, thereby decreasing the amount of Inferno tokens burned. This undermines the deflationary mechanism of the protocol, potentially impacting token value and user confidence.

The attack requires significant resources due to flashloan and transaction fees. The primary risk is the erosion of the protocol's economic incentives and market trust, though there is no direct economic gain for the attacker. The high cost associated with executing the attack every 28 minutes reduces its feasibility and the lack of a strong economic motive further limits its impact.

While the attack can disrupt the protocol's deflationary mechanism, the overall risk to the system is considered low.

Recommendation: Due to the permissionless nature of the protocol the issue is hard to be mitigated. Here are a couple of solutions:

1. Dynamic slippage protection based on TWAP - since TWAP oracles are hard to manipulate, a dynamic slippage protection based on TWAP oracle could be implemented to reliably configure the slippage protection. More on Uniswap V2 TWAPs: https://www.rareskills.io/post/twap-uniswap-v2
2. Another possibility is to add to the protocol slippage protection that is periodically configured manually by a trusted operator. Note that this solution requires trust in an operator/admin. This solution also introduces additional operational overhead as well as additional transaction costs.

Resolution: Acknowledged

[L-2] Inadequate slippage protection for Blaze/Inferno swaps

Context: InfernoBuyAndBurn.sol#L377

Description: The current implementation uses Uniswap V3's TWAP to calculate the minimum expected amount of Inferno when swapping Blaze for Inferno. In general this is a good approach. However, in the current instance the TWAP is based on the Blaze/Inferno UniV3 pool deployed by this protocol. To reduce the deployment gas fees, all UniswapV3 pools are deployed with an observation cardinality of 1. This means that the pool persists only the most recent tick. This effectively means that the pool would not present any TWAP functionality as-is which undermines slippage protection implemented here.

The severity of this finding is assessed as Low because the impact is very low given that the `swapTitanXForInfernoAndBurn()` is called every 28 minutes. Thus, it would be inefficient for an attacker to exploit this vulnerability to impact the Inferno burn rate in a major way.

Recommendation: Ensure that depending on the planned trading activity, an adequate observation cardinality is set for the Blaze/Inferno UniV3 pool via `IUniswapV3PoolActions#increaseObservationCardinalityNext()`. This will increase the number of slots that Uniswap V3 TWAP mechanism uses to store historical ticks.

Resolution: Fixed in d0d923d28814017d767452918334e91082c5fa1a

[L-3] Burned Inferno fees collected via the UniV3 pool are not tracked via `totalInfernoBurnt`

Context: InfernoBuyAndBurn.sol#L283, InfernoBuyAndBurn.sol#L248

Description: The `InfernoBuyAndBurn` contract has a state variable that tracks all Inferno that is burned: `totalInfernoBurnt`. In the current implementation, it only tracks Inferno that is burned via the buy-and-burn scheme. However, any Inferno fees collected via the UniV3 pool which are also burned are not tracked in it.

This may have unwanted consequences in terms of reporting if `totalInfernoBurnt` is used for reporting/stats purposes.

Recommendation: Make sure burned fees are accounted for in `totalInfernoBurnt`

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[L-4] Missing `_startTimestamp` validation

Context: InfernoBuyAndBurn.sol#L142

Description: As per the protocol whitepaper, the mint cycles should start on Firday 2pm UTC. The `_startTimestamp` that is used as the start time for the initial mint cycle is not validated and could be set to any weekday and time.

Recommendation: Since this aspect is related to the tokenomics of the protocol, it is recommended to add validation for `_startTimestamp`. E.g.

```
bool is2pmUTC = (_startTimestamp % 1 days) == 14 hours;
bool isFriday = ((blockTimestamp / 1 days + 4) % 7) == 5;
require(is2pmUTC && isFriday);
```

Resolution: Acknowledged

[L-5] Calculation will overflow and `getCurrentMintCycle()` will start from 1 again every 5 years

Context: InfernoMinting.sol#L169

Description: An unsafe downcast to `uint8` will overflow every 255 weeks (or ~5 years) since `type(uint8).max` is 255. Due to a check in the `mint()` function which checks the mint cycle `endDate` this cannot be exploited and is thus assessed as an informational finding.

Recommendation: Ensure sufficient checks are in place when downcasting or choose a bigger uint data type.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[L-6] Inconsistent intervals offset and count due to non-divisibility of 1440 minutes by 28 minutes

Context: InfernoBuyAndBurn.sol#L461

Description: The current implementation uses intervals of 28 minutes. However, 1440 minutes (24 hours) is not divisible by 28 minutes, causing an offset when wrapping around to a new day. This results in intervals starting at different times each day.

For example, if intervals start at 00:00 UTC on the first day, they will start at 00:16 UTC on the second day, calculated as `28 - (1440 % 28)`.

Consequently, some days will have fewer intervals than required. Specifically, the second day will have 50 intervals instead of the required 51, as calculated below:

```
(1440 - 16) / 28 = 50
```

Recommendation: Unless there is a specific need for intervals to be exactly 28 minutes, consider using an interval length that is a divisor of 1440, e.g. 30. This will ensure consistent interval start times and an equal and correct number of intervals per day.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[L-7] Missed intervals burn rate is calculated incorrectly

Context: InfernoBuyAndBurn.sol#L403

Description: When there are missed intervals for a couple of days, the missed intervals burn rate is calculated incorrectly. The current implementation backfills the burn allocation for the current interval to all missed intervals which leads to a different burn rate from the one described in the whitepaper.

Example scenario:

1. The last buy-and-burn was on Sunday (4% burn rate per day)
2. No one calls the buy-and-burn mechanism till Friday
3. Alice calls the buy-and-burn mechanism on Friday (15% burn rate per day)

Expected behavior: all missed intervals will be filled with their corresponding burn rate depending on their weekday

Actual behavior: all missed intervals are backfilled with the burn rate of the current weekday, i.e. 15% in the described scenario

The likelihood of this issue occurring is extremely low since users are incentivized to regularly call the buy-and-burn mechanism because they receive a portion of the tokens. Thus, despite the impact of this finding being high due to its extremely low likelihood the severity is low.

Recommendation: Make sure all missed intervals use the burn rate for their corresponding weekday.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

Informational

[I-1] `startTimestamp` should be marked as immutable

Context: InfernoMinting.sol#L52

Description: The `startTimestamp` state variable is only set in the constructor. Its value is never changed after that. It is recommended to mark it as `immutable` as this would lead to better gas efficiency.

When marking a variable as `immutable` it becomes part of the smart contract bytecode and does not occupy the smart contract storage. This leads to better gas efficiency since `SLOAD` is not used when reading its value.

Recommendation: Mark the `startTimestamp` state variable as immutable

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[I-2] Unreliable method for setting initial mint cycle start time

Context: DeployInferno.sol#L19

Description: Using `block.timestamp + 2 days` here implies that the contract will be deployed precisely at 2pm UTC 2 days before the first mint cycle. This is unreliable because deployment timing cannot be guaranteed due to external factors (e.g. network congestion). Although highly unlikely, a malicious actor could also delay the transaction via block stuffing. It's better to use a fixed timestamp to ensure the mint cycle starts exactly at 2pm UTC.

Alternatively, if a relative timestamp must be used, consider using `((block.timestamp / 1 days) * 1 days) + (2 days + 14 hours)` to set the first mint cycle 2 days from the transaction execution at exactly 2pm UTC.

The impact here is that if the start time of the initial minting cycle is delayed, this will also offset all subsequent minting cycles.

Recommendation: Use a concrete unix timestamp to have full control of the mint cycle start time

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[I-3] Code Improvements

Context: InfernoMinting.sol#L173, InfernoBuyAndBurn.sol#L47

Description:

1. The implementation could be simplified as follows:

```
startsAt = startTimestamp + ((currentCycle - 1) * GAP_BETWEEN_CYCLE)
```

2. Consider using `bool` instead of `uint8` for boolean flags. It is of the same size (8 bits) and uses `uint8` under the hood. However, the code would be simplified and more readable when using

bool:

```
uint8 isBlazeToken0;
```

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[I-4] Misleading parameter name

Context: InfernoBuyAndBurn.sol#L205

Description: The parameter name here is named `_amountTitanMin` however it specifies the minimum amount of Blaze tokens we are willing to accept during the TitanX/Blaze swap.

Recommendation: Rename the parameter to `_amountBlazeMin`

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[I-5] The `NoOngoingCycle()` and `AlreadyMintedForLP()` errors are not used anywhere

Context: InfernoMinting.sol#L59, Inferno.sol#L26

Description: The `NoOngoingCycle()` and `AlreadyMintedForLP()` errors are not used anywhere in the contracts. If these are leftovers - remove them.

Recommendation: Remove the `NoOngoingCycle()` and `AlreadyMintedForLP()` errors if they are not used.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

[I-6] Constructor unnecessarily marked as `payable`

Context: Inferno.sol#L36

Description: The `Inferno` constructor is unnecessarily marked as `payable`. Marking the constructor as `payable` allows the contract to receive ETH during deployment. If this was not the intention, it could lead to situations where ETH is mistakenly sent to the contract during deployment.

The gas savings for marking the constructor as `payable` are also minuscule (~24 gas) and only saved during deployment time.

Recommendation: Remove the `payable` modifier from the `Inferno` contract constructor

Resolution: Acknowledged

[I-6] Incorrect/incomplete natspec

Context: InfernoMinting.sol#L78, Inferno.sol#L11, Inferno.sol#L31-L33

Description:

1. The natspec for the `InfernoMinting` constructor is outdated and incomplete.
2. The natspec for `Inferno` says INFERNO can only be minted by the `InfernoMinter` (should

be `InfernoMinting`) but tokens can also be minted by `InfernoBuyAndBurn` for the initial LP mint.

3. The natspec for the `Inferno` constructor says it sets the minter contract address. However, it also configures the burner address as well.
4. The `InfernoAuction` contract is an outdated name of `InfernoMinting` which needs to be updated.

Resolution: Fixed in 1c818b6f741292091ea31dbfaea2aa2c6f305458

4 Other Risks

Centralization Risk

The current implementation permits the owner of the `InfernoBuyAndBurn` and `InfernoMinting` contracts to modify their associated `Inferno` token address. This introduces a centralization risk, as the owner could potentially disrupt the buy-and-burn mechanism by assigning another token.

Recommendation: To ensure the protocol remains trustless, the owner should not be able to update the references to the Inferno token once they are set. There are a couple of options to achieve this:

- Use constructor dependency injection instead of setter dependency injection. This approach makes the `Inferno` contract the entry point during deployment, responsible for creating the `InfernoMinting` and `InfernoBuyAndBurn` contracts. By setting all token and pool addresses as immutable constants in the contracts, the risk is mitigated.
- Renounce ownership of the `InfernoMinting` and `InfernoBuyAndBurn` contracts in `Deploy_Inferno.sol` once they are fully deployed and configured, ensuring no further modifications can be made.