marchev

GoatX
Security Review

Version 1.1

Reviewed by:

Martin Marchev

November 26, 2024

Table of Contents

1 Introduction

1.1 About Martin Marchev

Martin Marchev is an independent security researcher specializing in Web3 security. With a proven track record in competitive audits and responsible vulnerability disclosures, he has contributed to the security of high-profile projects with over $1B in Total Value Locked (TVL).

For bookings and security review inquiries, reach out via Telegram, Twitter or Discord.

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

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

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 GoatX

GoatX is a hyper-deflationary meme-fi token that combines an initial minting phase, buy-and-burn mechanics and perpetual auctions following the minting period.

2.2 Synopsis

Project Name	GoatX
Project Type	ERC-20
Repository	goatx-contracts
Commit Hash	8b5bb924...b4a69459
Review Period	13 Nov 2024 - 25 Nov 2024

2.3 Issues Found

Severity	**Count**
Critical Risk	2
High Risk	2
Medium Risk	1
Low Risk	6
Informational	2
Total Issues	**13**

3 Findings

Critical Risk

[C-1] Complete failure of the buy-and-burn mechanics due to incorrect transfer amount

Context: Minting.sol#L270

Description: When a user deposits funds to the `GoatXMinting` contract, once sufficient liquidity is raised for LP bootstrapping and future liquidity provisioning, the remaining TITANX must be distributed according to the protocol's whitepaper. A significant portion of this distribution (38%) is intended to be sent to the `GoatXBuyAndBurn` contract to enable the buy-and-burn mechanism, a key component of the protocol's deflationary design.

However, due to the following implementation, only 0.38 TITANX tokens are transferred per deposit instead of 38% of the accumulated TITANX balance:

```
titanX.transfer(address(goatX.buyAndBurn()), uint256(0.38e18)); //@audit Transfers
↪  only 0.38 TITANX instead of 38% of the balance
```

This issue results in two critical problems:

1. **Breakdown of buy-and-burn mechanism:** The intended deflationary effect of the protocol is entirely compromised as the buy-and-burn contract receives negligible funds ($0.00000018).
2. **Locked TITANX Funds:** 38% of the TITANX balance remains trapped in the `GoatXMinting` contract. While it is technically possible to distribute the locked funds to other distribution constituents by repeatedly making small deposits (e.g. 1 wei of TITANX), this workaround is highly impractical and ineffective. Moreover, the buy-and-burn mechanism would remain unfulfilled, regardless of this approach.

Recommendation: Fix the implementation so that it transfers 38% of the TITANX balance instead of 0.38 TITANX:

```diff
@@ -267,7 +267,7 @@ contract GoatXMinting is SwapActions, IGoatXMinting {
             titanX.transfer(Constants.LP_WALLET, amountToAdd);
         }

-        titanX.transfer(address(goatX.buyAndBurn()), uint256(0.38e18));
+        titanX.transfer(address(goatX.buyAndBurn()), wmul(_amount, uint256(0.38e18)));

         {
             uint256 toAuctionBuy = wmul(_amount, uint256(0.3e18));
```

Resolution: Fixed in 2410883

[C-2] All GoatX tokens bought by `AuctionBuy` get irrecoverably trapped due to missing transfer to `GoatFeed`

Context: AuctionBuy.sol#L135-L138

Description: According to the whitepaper, the `AuctionBuy` contract should use the accumulated TitanX to purchase GoatX from the market and transfer the GoatX tokens to the `GoatFeed` contract, which then feeds the `Auction`mechanism.

However, the implementation of `swapTitanXToGoatXAndFeedTheAuction()` contains a critical flaw. The function transfers the incentive fee to the calling user and swaps the remaining allocated TitanX for GoatX. However, it does not transfer the purchased GoatX to the `GoatFeed` contract as intended.

As a result, all purchased GoatX tokens remain permanently trapped in the `AuctionBuy` contract without any possibility to recover them. This disrupts the protocol's tokenomics and renders the auction mechanism inoperative as the `GoatFeed` contract is never supplied with GoatX.

Recommendation: Fix the `swapTitanXToGoatXAndFeedTheAuction()` function to ensure that the purchased GoatX tokens are transferred to the `GoatFeed` contract:

```
uint256 goatXAmount =
    swapExactInput(address(titanX), address(goatX), currInterval.amountAllocated -
     ↪  incentive, 0, _deadline);

goatX.safeTransfer(goatX.goatFeed(), goatXAmount);
```

Resolution: Fixed in 2410883

High Risk

[H-1] Loss of user funds in case of multiple deposits per minting cycle

Context: Minting.sol#L131-L132

Description: If a user makes multiple deposits during an auction cycle, only the data from their last deposit will be persisted. The current implementation overwrites any existing deposit data for the cycle:

```
deposits[msg.sender][currentCycle] =
    Deposit({depositedAt: uint32(block.timestamp), titanXAmount: uint216(_amount),
     ↪  cycle: currentCycle});
```

This logic results in the loss of all previously deposited funds for the user in the same cycle.

Recommendation: Allow users to make multiple deposits within the same cycle by accumulating deposit data for each user. This can be achieved by modifying the implementation to sum the `titanXAmount` for the current minting cycle.

Resolution: Fixed in 2410883

[H-2] GoatX/TitanX liquidity bootstrapping vulnerable to griefing via empty pool price manipulation

Context: Minting.sol#L238-L239

Description: The `addInitialLiquidity()` function in the GoatX protocol bootstraps liquidity in the GoatX/TitanX UniswapV3 pool once $5000 worth of TitanX is raised. The protocol employs a hardcoded slippage protection of 80%, which determines the `amount0Min` and `amount1Min` parameters for the liquidity minting process.

Empty UniswapV3 pools allow zero-cost swaps that push the pool price to the `MIN_TICK` or `MAX_TICK`. A malicious actor can exploit this characteristic to manipulate the price in the GoatX/TitanX pool. The manipulated price causes the slippage protection to revert the transaction, preventing the liquidity minting. Additionally, the attacker can add single-sided TitanX liquidity in a price range above the manipulated price but below the intended price. This exacerbates the issue, as GoatX tokens (unavailable until the second day) are required to restore the intended price.

If the protocol admin attempts to restore the price using GoatX tokens minted to the LP wallet (violating the whitepaper), the attacker can escalate the attack by adding more TitanX liquidity, making the required GoatX amount infeasible.

This vulnerability enables a malicious actor to disrupt or delay the liquidity provisioning of the GoatX/TitanX Uniswap v3 pool.

Recommendation: To mitigate this vulnerability, deploy the GoatX token programmatically only after sufficient TitanX is raised and execute liquidity provisioning along with the deployment atomically in a single transaction. This approach ensures that no Uniswap v3 pool can be created with GoatX before sufficient TitanX is raised, preventing price manipulation in an empty pool.

Resolution: Fixed in 2410883

Medium Risk

[M-1] Perpetual auctions start before minting phase completes

Context: DeployGoatX.s.sol#L34

Description: The current configuration of the `AUCTION_START_TIME` sets the perpetual auctions to begin 12 days after the GoatX minting starts:

```
uint32 AUCTION_START_TIME = uint32(MINTING_START_TIME + 12 days);
```

However, according to the whitepaper, the perpetual auctions must commence only after the minting phase is complete:

> *Once the minting period ends GOATX becomes capped and deflationary, the perpetual auctions begin.*

The minting phase consists of 14 cycles, with each cycle lasting 1 day. This means the minting phase lasts 14 days, whereas the current implementation starts perpetual auctions prematurely on the 12th day. This discrepancy alters the intended protocol tokenomics by overlapping the minting phase with the auction phase.

This may potentially leading to economic disruption as the protocol transitions to perpetual auctions while the minting phase is still ongoing, changing the intended supply dynamics of the GOATX token. Moreover, participants expecting auctions to start after minting completes will be misled and may miss out on the auction opportunities.

Recommendation: Adjust the `AUCTION_START_TIME` to begin after the full 14-day minting phase. The corrected implementation should look like this:

```diff
@@ -31,7 +31,7 @@ contract DeployGoatX is DeployConfig {

         uint32 MINTING_START_TIME = uint32(1732802400);
         uint32 AUCTION_BUY_START = uint32(MINTING_START_TIME + 1 days);
-        uint32 AUCTION_START_TIME = uint32(MINTING_START_TIME + 12 days);
+        uint32 AUCTION_START_TIME = uint32(MINTING_START_TIME + 14 days);

         goatX = new GoatX(c.titanX, c.v3PositionManager);
```

This change ensures alignment with the whitepaper's specifications and preserves the intended tokenomics of the protocol.

Resolution: Fixed in 2410883

Low Risk

[L-1] Overflow in `_calculateMissedIntervals()` after prolonged inactivity

Context: AuctionBuy.sol#L258

Description: The `_calculateMissedIntervals()` function calculates the number of missed intervals as follows:

```
function _calculateMissedIntervals(uint256 timeElapsedSince) internal view returns
 ↪  (uint16 _missedIntervals) {
    _missedIntervals = uint16(timeElapsedSince / INTERVAL_TIME);

    if (lastBurnedIntervalStartTimestamp != 0) _missedIntervals--;
}
```

The `_missedIntervals` variable is a `uint16` and will overflow for values greater than 65535. This equates to a maximum of 65535 intervals or 524,280 minutes (or approximately 364,083 days). If the contract remains inactive beyond this threshold, the calculation will silently overflow, leading to incorrect accounting of missed intervals.

While the likelihood of this scenario is very low — given users are incentivized to interact with the contract — it remains theoretically possible and could disrupt the contract's functionality in edge cases.

Recommendation: Use a larger data type, such as `uint32`, for `_missedIntervals` to prevent overflow and allow for significantly larger inactivity periods.

Resolution: Fixed in 2410883

[L-2] Ineffective default `swapCap` value in `BuyAndBurn` and `AuctionBuy` contracts

Context: AuctionBuy.sol#L71, BuyAndBurn.sol#L34

Description: The `BuyAndBurn` and `AuctionBuy` contracts use a default `swapCap` value of `type(uint128).max`, which is ineffective at preventing MEV sandwich attacks. This value essentially allows unlimited swaps, creating a potential vulnerability where MEV bots can exploit large transactions to drain value from the protocol.

Additionally, the deployment scripts do not configure a sensible default value for `swapCap` during deployment. This increases the risk that the `swapCap` remains set to `type(uint128).max` post-deployment due to oversight, exposing the protocol to MEV exploitation.

Recommendation: Add `swapCap` as a mandatory parameter in the contract constructors. This will enforce proper configuration during deployment and ensure that the `swapCap` value is set to a sensible limit.

Resolution: Acknowledged

[L-3] Incorrect argument passed to `UserDeposit` event

Context: Auction.sol#L84

Description: The `Auction.deposit()` function emits a `UserDeposit` event when a user deposits funds into the auction. Each user deposit is assigned a unique ID:

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

The `UserDeposit` event is defined as:

```
event UserDeposit(address indexed user, uint256 indexed amount, uint64 indexed id);
```

However, the function incorrectly passes `daySinceStart` as the third argument to the `UserDeposit` event instead of the expected `depositId`:

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

This results in incorrect information being emitted, which could mislead off-chain tracking systems and developers relying on the event data.

Recommendation: Emit the `UserDeposit` event with the correct `depositId` parameter:

```
@@ -81,7 +81,7 @@ contract GoatXAuction is Errors {

         _distribute(_amount);

-        emit UserDeposit(msg.sender, _amount, daySinceStart);
+        emit UserDeposit(msg.sender, _amount, depositId);
     }

     function claim(uint64 _id) public {
```

Resolution: Fixed in 2410883

[L-4] Incorrect event emitted in `swapTitanXToGoatXAndFeedTheAuction()`

Context: AuctionBuy.sol#L142

Description: The `swapTitanXToGoatXAndFeedTheAuction()` function in the `AuctionBuy` contract emits a `BuyAndBurn()` event, which incorrectly suggests that the function facilitates a buy-and-burn operation. However, the function's actual purpose is to swap TitanX for GoatX and feed the auction mechanism. This mismatch can lead to confusion and cause off-chain systems relying on protocol events to process incorrect data.

Recommendation: Emit an appropriately named event to reflect the function's actual behavior. For example:

```
// Define the new event
event AuctionFed(uint256 goatXAmount);

// Emit the correct event in the function
emit AuctionFed(goatXAmount);
```

Resolution: Fixed in 2410883

[L-5] Missing event emission in `GoatXMinting.deposit()`

Context: Minting.sol#L116

Description: The `deposit()` function in the `GoatXMinting` contract is a state-changing function that processes user deposits. It is a common best practice for important and state-changing functions to emit events, as events enable off-chain tracking and improve transparency for users and developers.

However, the `deposit()` function does not currently emit any events. This omission makes it harder to monitor deposits and track protocol activity through off-chain tools.

Recommendation: Introduce an event for the `deposit()` function, such as `Deposit()`, and emit it whenever the function is called.

Resolution: Fixed in 2410883

[L-6] Resolve all TODOs

Context: Constants.sol#L5-L11

Description: The codebase contains several TODO comments, including in `Constants.sol`, where placeholder address constants are used. Some of these address constants are of critical importance for the protocol's functionality and security, such as those defining key contracts or wallet addresses.

Leaving these TODOs unresolved can result in deployment errors if placeholder values are left unchanged or potential vulnerabilities if incorrect addresses are used in production.

Recommendation: Review and resolve all TODO comments in the codebase before deployment. Pay particular attention to address constants in `Constants.sol` and ensure they are assigned correct values before the mainnet deployment takes place.

Resolution: Acknowledged

Informational

[I-1] The non-EOA caller protection will become ineffective with EIP-7645

Context: AuctionBuy.sol#L92, Errors.sol#L37

Description: The `onlyEOA` modifier is used in the project to prevent non-EOA interactions with particular functions:

```
modifier onlyEOA() {
    require(msg.sender.code.length == 0 && tx.origin == msg.sender, OnlyEOA());
    _;
}
```

The first condition can easily be circumvented by calling the function from within a new contract constructor since during contract construction `code.length` is equal to `0`.

The second condition ensures that `msg.sender` is equal `tx.origin` which is only true for EOAs. However, if EIP-7645 gets accepted, `msg.sender` and `tx.origin` would become aliases, rendering this check ineffective.

Recommendation: Contracts should not rely on the origin of the transactions as a protection mechanism. Instead, implement other suitable mechanisms depending on the context, such as TWAP-based protections, swap caps, cooldown periods, etc. These approaches provide more robust protection and are not impacted by the changes proposed in EIP-7645.

Resolution: Acknowledged

[I-2] Add validation for `startTimestamp` in `GoatXMinting`

Context: Minting.sol#L61

Description: The `_startTimestamp` parameter in the `GoatXMinting` constructor determines the starting point of minting cycles. According to the protocol's whitepaper, these cycles must begin at 2PM UTC precisely. This timestamp also defines the cutoff time for each minting cycle. Ensuring this precision is critical for the proper functioning of the project's minting mechanism.

Currently, there is no validation to ensure that the `_startTimestamp` indeed corresponds to 2PM UTC. An erroneous configurationwould disrupt the alignment of the minting process with the expected schedule, potentially affecting user experience and protocol integrity.

Recommendation: Add validation to confirm that the `_startTimestamp` passed to the contract denotes 2PM UTC on the specified date. This can be achieved by checking that the timestamp aligns with the desired hour and minute values in UTC:

```
require((_startTimestamp % 86400) == 50400, "_startTimestamp must be 2PM UTC");
```

Resolution: Fixed in 2410883