Goat Trading Audit Report

Introduction

An internal security review of the **Goat Trading protocol** alongside protocol development was done by **0xanmol**, with a focus on the security aspects of the application's smart contracts 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 I try to find as many vulnerabilities as possible. I 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 0xanmol

Anmol Pokhrel, or **0xanmol**, is an independent smart contract security researcher. Having a deep interest in Web 3 security, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews.

Twitter

Github

About Goat Trading

Goat Trading is a phased Automated Market Maker (PAMM) protocol that enables anyone to launch their token pool to accumulate a specific amount of WETH. Initially, this pool is known as a presale AMM and includes a virtual amount in the reserve. As swaps gradually increase the desired WETH amount, the presale pool changes into an x*y AMM. The initial Liquidity provider then owns all the raised WETH. Besides token launch, Goat Trading also features built-in MEV protection and allows LPs to withdraw rewards in WETH without removing liquidity.

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

Repo: https://github.com/inedibleX/goat-trading

The security review was carried out simultaneously with the development of the protocol, thus there are no specific commit hashes. The issues were discovered across multiple commits.

Scope

- exchange
  - GoatV1pair.sol

    - GoatV1ERC20.sol
    - Goatv1Factory.sol
- periphery
    - GoatV1Router.sol
- library
    - GoatTypes.sol
    - GoatLibrary.sol

Findings Summary

ID	Title	Severity	Status
[C-01]	Initial LP can bypass fractional liquidity checks by passing different `to` address when removing liquidity.	Critical	Fixed
[C-02]	Lp fees addition to reserves in presale phase misstep in swap function halts transition to presale endangering pool funds	Critical	Fixed
[H-01]	Unfair token distribution to the LPs on liquidity burn.	High	Fixed
[M-01]	Anyone holding the token can create a pool with a dust amount and convert that to amm, not giving the actual team a chance to raise WETH	High	Acknowledged
[M-02]	An attacker could potentially reduce the fractional liquidity of the initial LP by minting a dust amount.	Medium	Acknowledged
[M-03]	Unnecessary state changes occur when calling `_updateInitialLpInfo` with `isBurn` set as true within `_burnLiquidityAndConvertToAmm`	Medium	Acknowledged
[M-04]	Anyone can grief the user wanting to sell his token during the vesting period.	Medium	Acknowledged

[L-01]	Insufficient data is passed when emitting events for `mint` and `burn`	Medium	Acknowledged
[G-01]	Reading the state only when necessary can help save gas in the mint function.	Low	Acknowledged
[G-02]	`updateFeeReward` can be skipped if `to` address is `pair` in `_beforeTokenTransfer`	Low	Acknowledged

[C-01] Initial LP can bypass fractional liquidity checks by passing different `to` address when removing liquidity.

Impact: High, as the protocol will compromise its core functionality to prevent token dumping

Likelihood: High, as teams can directly dump their all tokens.

Description

The `to` address given in a burn function is used to determine if that address is the initial liquidity provider (LP). If the `to` address matches the initial LP, then fractional liquidity logic is implemented to prevent the initial LP from dumping all the tokens and walking away with the raised WETH.

However, the initial LP can bypass this check simply by providing a different `to` address than the one recorded as the initial LP.

Recommendation

The protocol should be able to identify if the caller of the burn function is the initial Liquidity Provider (LP). One approach could be to store the address that transfers

the LP token to the pair contract before invoking the burn function. before transfer hook can be used to achieve this.

[C-02] Lp fees addition to reserves in presale phase misstep in swap function halts transition to presale endangering pool funds

Impact: High, as it can cause complete DOS of the pool.

Likelihood: High. This will happen almost every time.

Description

In the swap function, there is a check to determine whether the presale pool can be converted to an AMM using the provided WETH input. This check occurs after subtracting the 99 basis points (BPS) fees from the input WETH.

```
if (swapVars.isPresale && swapVars.finalReserveEth >= swapVars.l
            // at this point pool should be changed to an AMM
            _checkAndConvertPool(swapVars.finalReserveEth, swapV
        }
```

In the presale phase, the `liquidityFees` are added to the reserves instead of distributed to the initial LP. The following check is performed to do so at the very last of the swap function just before calling `update`

```
 if (swapVars.isPresale) {
    // presale lp fees should go to reserve eth
     swapVars.finalReserveEth += swapVars.lpFeesCollected;
   }
```

Currently, we are excluding the `liquidityFees` from the reserves when checking if the pool can be converted to an AMM. However, it is possible that adding those fees can enable the conversion to AMM. This leads to a violation of the protocol's invariant, where the pool remains in a presale state even though the real WETH reserve exceeds the bootstrap amount.

This incorrect state will cause a `getTokenAmountOutPresale` to underflow, resulting in the protocol getting stuck in the presale phase and causing the loss of all user funds.

Impact

The pool will encounter a Denial of Service (DOS) attack, which will result in the funds of users becoming inaccessible.

Recommendation

Instead of adding `liquidityFees` at the very end of the swap function,it should be done before checking the final reserve with bootstrap WETH to convert the pool from presale to AMM. The code swap function should looks like this after the update.

```
function swap(
        uint256 amountTokenOut,
        uint256 amountWethOut,
        address to
    ) external nonReentrant {
        if (amountTokenOut == 0 && amountWethOut == 0) {
            revert GoatErrors.InsufficientOutputAmount();
        }
        if (amountTokenOut != 0 && amountWethOut != 0) {
            revert GoatErrors.MultipleOutputAmounts();
        }
        GoatTypes.LocalVariables_Swap memory swapVars;
        swapVars.isBuy = amountWethOut > 0 ? false : true;
        // check for mev
        _handleMevCheck(swapVars.isBuy);
```

```
        (
            swapVars.initialReserveEth,
            swapVars.initialReserveToken
        ) = _getActualReserves();

        if (
            amountTokenOut > swapVars.initialReserveToken ||
            amountWethOut > swapVars.initialReserveEth
        ) {
            revert GoatErrors.InsufficientAmountOut();
        }

        if (swapVars.isBuy) {
            swapVars.amountWethIn =
                IERC20(_weth).balanceOf(address(this)) -
                swapVars.initialReserveEth -
                _pendingLiquidityFees -
                _pendingProtocolFees;
            // optimistically send tokens out
            IERC20(_token).safeTransfer(to, amountTokenOut);
        } else {
            swapVars.amountTokenIn =
                IERC20(_token).balanceOf(address(this)) -
                swapVars.initialReserveToken;
            // optimistically send weth out
            IERC20(_weth).safeTransfer(to, amountWethOut);
        }
        swapVars.vestingUntil = _vestingUntil;
        swapVars.isPresale = swapVars.vestingUntil == _MAX_UINT3

        (swapVars.feesCollected, swapVars.lpFeesCollected) = _ha
            swapVars.amountWethIn,
            amountWethOut,
            swapVars.isPresale
        );
```

```
        swapVars.tokenAmount = swapVars.isBuy
            ? amountTokenOut
            : swapVars.amountTokenIn;

        // We store details of participants so that we only allo
        // swap back tokens who have bought in the vesting perio
        if (swapVars.vestingUntil > block.timestamp) {
            _updatePresale(to, swapVars.tokenAmount, swapVars.is
        }

        if (swapVars.isBuy) {
            swapVars.amountWethIn -= swapVars.feesCollected;
        } else {
            unchecked {
                amountWethOut += swapVars.feesCollected;
            }
        }
        swapVars.finalReserveEth = swapVars.isBuy
            ? swapVars.initialReserveEth + swapVars.amountWethIn
            : swapVars.initialReserveEth - amountWethOut;
        swapVars.finalReserveToken = swapVars.isBuy
            ? swapVars.initialReserveToken - amountTokenOut
            : swapVars.initialReserveToken + swapVars.amountToke

        swapVars.bootstrapEth = _bootstrapEth;
        + if (swapVars.isPresale) {
        +    // presale lp fees should go to reserve eth
        +    swapVars.finalReserveEth += swapVars.lpFeesCollecte
            // at this point pool should be changed to an AMM
            if (swapVars.finalReserveEth >= swapVars.bootstrapEt
                _checkAndConvertPool(
                    swapVars.finalReserveEth,
                    swapVars.finalReserveToken
                );
            }
        } else {
```

```
            // check for K
            swapVars.initialTokenMatch = _initialTokenMatch;
            swapVars.virtualEth = _virtualEth;

            (
                swapVars.virtualEthReserveBefore,
                swapVars.virtualTokenReserveBefore
            ) = _getReserves(
                swapVars.vestingUntil,
                swapVars.initialReserveEth,
                swapVars.initialReserveToken
            );
            (
                swapVars.virtualEthReserveAfter,
                swapVars.virtualTokenReserveAfter
            ) = _getReserves(
                swapVars.vestingUntil,
                swapVars.finalReserveEth,
                swapVars.finalReserveToken
            );

            if (
                swapVars.virtualEthReserveBefore *
                    swapVars.virtualTokenReserveBefore >
                swapVars.virtualEthReserveAfter *
                    swapVars.virtualTokenReserveAfter
            ) {
                revert GoatErrors.KInvariant();
            }
        }
-  if (swapVars.isPresale) {
- // presale lp fees should go to reserve eth
- swapVars.finalReserveEth += swapVars.lpFeesCollected;
-      }

        _update(swapVars.finalReserveEth, swapVars.finalReserveT
```

```
        // TODO: Emit swap event with similar details to uniswap
        // @note what should be the swap amount values for emit
    }
```

[H-01] Unfair token distribution to the LPs on liquidity burn.

Impact: High, as the initial burner can get more WETH.

Likelihood: High, LP's would always do this to get the profit, eventually breaking the fees mechanics.

Description

When burning liquidity, the amount of WETH lp a user should receive is calculated based on their share of the pool's WETH balance.

```
        uint256 balanceEth = IERC20(_weth).balanceOf(address(thi
        uint256 balanceToken = IERC20(_token).balanceOf(address(

        uint256 totalSupply_ = totalSupply();
        amountWeth = (liquidity * balanceEth) / totalSupply_;
        amountToken = (liquidity * balanceToken) / totalSupply_;
```

The balance of WETH in a contract consists of the sum of reserves, lpFees, and protocolFees. When we calculate the WETH using the balances, the unclaimed fees are also included. This will lead to more WETH being distributed to the lp's who burn before anyone claims the fees. Further, this system may face issues in the long run if everyone starts claiming fees, as there will be no fees available for everyone.

Impact

1. Unfair WETH distribution to LP's
2. The fee mechanics will break

Recommendation

Use reserves instead of balances.

```
        uint256 balanceEth = IERC20(_weth).balanceOf(address(thi
        uint256 balanceToken = IERC20(_token).balanceOf(address(

        uint256 totalSupply_ = totalSupply();
      - amountWeth = (liquidity * balanceEth) / totalSupply_;
      - amountToken = (liquidity * balanceToken) / totalSupply_
      + amountWeth = (liquidity * _reserveEth) / totalSupply_;
      + amountToken = (liquidity * _reserveToken) / totalSupply
```

[M-01] Anyone holding the token can create a pool with a dust amount and convert that to amm, not giving the actual team a chance to raise WETH.

Impact: High, since this could prevent the team from raising any WETH.

Likelihood: Medium, because the attacker would need to hold the team token and there's no direct financial incentive for them.

Description

An attacker with the token could front-run the actual pool creation by creating a pool with a dust amount and converting it directly to AMM. This could be easily

accomplished by providing a very small `bootstrapEth` and an equal `initialEth` amount.

Once the pool is created and converted to AMM, the actual team cannot use the `takeOver` function to gain control of the pool.

Recommendation

Prevent `bootstrapEth` from being a dust amount and establish a minimum requirement for bootstrapEth.

For instance, if the protocol only permits pool creation with 1 WETH, an attacker would need to put 1 WETH to convert it directly to AMM, which wouldn't be beneficial for them.

[M-02] An attacker could potentially reduce the fractional liquidity of the initial LP by minting a dust amount.

Impact: High. The system might not function as expected for the initial LP.

Likelihood: Low, as there's no direct benefit for the attacker.

Description

If a malicious user sets a `to` address as the initial Liquidity Provider (LP) in the `mint` function, it alters the fractional liquidity and remaining withdrawal for the initial LP, potentially decreasing the liquidity that the initial LP could withdraw.

For example, suppose the initial LP has a liquidity balance of 75 and a remaining withdrawal of 3, indicating a fractional liquidity of 25.

If a malicious actor mints a liquidity of 1 to this LP, the remaining withdrawal resets to 4, and the fractional liquidity changes to 76 / 4 = 19.

If an attacker repeats this action, the initial LP may never be able to withdraw all of their amounts, even after the vesting duration ends.

```
 if (mintVars.isFirstMint || to == _initialLPInfo.liquidityProvi
      _updateInitialLpInfo(liquidity, balanceEth, to, false, fal
    }
```

```
function _updateInitialLpInfo(
    uint256 liquidity,
    uint256 wethAmt,
    address lp,
    bool isBurn,
    bool internalBurn
  ) internal {
    GoatTypes.InitialLPInfo memory info = _initialLPInfo;

    if (internalBurn) {
      // update from from swap when pool converts to an amm
      info.fractionalBalance = uint112(liquidity) / 4;
    } else if (isBurn) {
      if (lp == info.liquidityProvider) {
        info.lastWithdraw = uint32(block.timestamp);
        info.withdrawalLeft -= 1;
      }
    } else {
    //@audit decrease fractional balance of initial LP
      info.fractionalBalance = uint112(((info.fractionalBalance
      info.withdrawalLeft = 4;
      info.liquidityProvider = lp;
      if (wethAmt != 0) {
        info.initialWethAdded = uint104(wethAmt);
      }
    }

    // Update initial liquidity provider info
```

```
    _initialLPInfo = info;
  }
```

Recommendation

Prohibit the minting of liquidity to the initial LP.

[M-03] Unnecessary state changes occur when calling `_updateInitialLpInfo` with `isBurn` set as true within `_burnLiquidityAndConvertToAmm`

Impact: Medium. The initial liquidity provider can now withdraw their liquidity in 3 weeks instead of 2.

Likelihood: High. This will happen every time the pool converts to AMM.

Description

The `_updateInitialLpInfo` function serves to update the details of the initial liquidity provider (LP). It modifies their fractional balance and remaining withdrawals. If the initial LP is burning their liquidity, the number of remaining withdrawals decreases by one, and the last withdrawal timestamp is updated to the current time. This change is determined by the `isBurn` parameter passed into the function.

This function performs as expected for standard liquidity withdrawals. However, during the conversion of the presale pool to AMM, if the initial LP's liquidity is burned, it unnecessarily alters the remaining withdrawals and the last withdrawal time.

Consequently, the initial LP only needs to wait 3 weeks to withdraw all their liquidity.

Recommendation

In the `_updateIntialLpInfo` function, add a check to distinguish between internal burn and actual burn. If the internal burn is happening just calculate new fractional liquidity for LP.

[M-04] Anyone can grief the user wanting to sell his tokens during the vesting period.

Impact: High. The system might not function as expected for the user in the vesting phase.

Likelihood: Low, as there's no direct benefit for the attacker.

Description

After converting the presale pool into an AMM, a vesting period of 30 days is established. During this time, tokens can only be sold if they were purchased on the same curve. To keep track of user balances during the vesting period, the `_presaleBalances` mapping is utilized. Each time a user buys a token, their presale token balance increases, and when they sell the token, their presale balance decreases.
If users do not have a presale balance and attempt to sell tokens, it will result in an underflow, which is an expected functionality.

```
function _updatePresale(address user, uint256 amount, bool isB

      if (isBuy) {
          unchecked {
              _presaleBalances[user] += amount;
          }
      } else {
          _presaleBalances[user] -= amount;
      }
  }
```

The `uint256 amount` here is supposed to be the user-transferred token when doing a swap. It is calculated in the swap function with this code as `swapVars.amountTokenIn`

```
if (swapVars.isBuy) {
          swapVars.amountWethIn =
              IERC20(_weth).balanceOf(address(this)) -
              swapVars.initialReserveEth -
              _pendingLiquidityFees -
              _pendingProtocolFees;
          // optimistically send tokens out
          IERC20(_token).safeTransfer(to, amountTokenOut);
      } else {
          //@audit anyone can frontrun this and cause an arith
          swapVars.amountTokenIn =
              IERC20(_token).balanceOf(address(this)) -
              swapVars.initialReserveToken;
          // optimistically send weth out
          IERC20(_weth).safeTransfer(to, amountWethOut);
      }
```

Anyone can front-run the user. transferring 1 WEI and preventing the user from selling their tokens.

Flow

- swapVars.intialReserveToken = 500
- user presale balance in mapping = 500

  - attacker front-run and transfer 1 wei token in pair
- The user transfers 500 tokens to sell
- The balance of token became 500 + 500 + 1 = 1001
- Balance - intialReserve → 1001 - 500 = 5001

- The system assumes that the user has sent 501 tokens to sell
- The 501 is passed to `_updatePresale` to decrement the user's presale balances.
- This will underflow because the user only has 500 presale balance and 501 is subtracted from 500.

Impact

The system will not function as intended for swappers in the vesting phase.

Recommendation

It would be helpful to have a sync function that automatically synchronizes the balances with the reserves to minimize the likelihood of any errors. If such an error does occur, the transaction will be reverted for the user. They can then use the sync function to try again. However, it's important to note that this is not a complete solution to the problem.

[L-01] Insufficient data is passed when emitting events for `mint` and `burn`

Description

In the event data, `msg.sender` is passed as a user address inside the `mint` and `burn` functions. These functions are expected to be called by the router, so in this scenario, `msg.sender` will be a router.

Recommendation

add one more parameter in the event and pass `to` address there.

Gas Reports

[G-01] Saving gas in the mint function by reading the state only when necessary

Description

Reading the state can incur 2100 gas if it's cold storage and 100 gas if it's hot storage.

Within the `mint` function, certain states are required only when the pool is in a presale.

For example:

```
    //@audit this state read can be done inside the if block
     mintVars.virtualEth = _virtualEth;
     mintVars.initialTokenMatch = _initialTokenMatch;
     mintVars.bootstrapEth = _bootstrapEth;
```

Recommendation

Rather than always initializing these local variables by reading storage, it would be more gas-efficient to initialize them only when the pool is in a presale.

[G-02] `updateFeeRewards` can be skipped if `to` address is `pair` in `_beforeTokenTransfer`

Description

Inside `_beforeTokenTransfer` the rewards are updated for the sender and receiver of the token.

If the receiver is pair address itself then, there is no need to update the receiver's rewards. This can save a significant amount of gas for a sender.

```
 // Update fee rewards for both sender and receiver
   _updateFeeRewards(from);
   // @audit do not update fee if to is address(this)
   _updateFeeRewards(to);
```

Recommendation

Add a condition to check for pair address

```
// Update fee rewards for both sender and receiver
   _updateFeeRewards(from);
   if(to != address(this){
   _updateFeeRewards(to);
   }
```