Introduction

A smart contract security audit review of the Goat Trading was done by 0xladboy233, with a focus on the security aspects of the application's implementation.

Background from auditor

Ladboy233 is a seasoned blockchain security researcher, specializing in DeFi/blockchain security. His portfolio includes comprehensive audits for notable projects such as Optimism, Notional Finance, and The Graph Protocol. For a detailed overview of his security reviews and research, visit his GitHub repository at: https://github.com/JeffCX/Sparkware-audit-portfolio/.

Seeking to ensure the highest standards of security, the client turned to Sherlock, a reputable platform known for hosting audit contests and identifying critical vulnerabilities, to act as an intermediary in facilitating the audit process

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. We 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 **Protocol Name**

Goat trading allows teams to bootstrap their liquidity pools so no matching Ether is required to launch a pool, and gives users security far beyond what other dexes can provide. This dual focus on both teams and users ensures all actors in the ecosystem have the motivation required to use a new protocol. Despite the innovation within the protocol, we also maintain the trading style and algorithms that teams and users are used to to in order to make a transition from the traditional and unsafe trading style as easy as possible.

Finding summary:

High - 1
Medium - 7
Low - 2

H1 – User account A can maliciously mint LP token to account B to block account B's LP token transfer

Vulnerability Detail

When user mint LP token to an account, the LP lock time for LP token recipient is extended.

```
function _mint(address _to, uint256 _value) internal {
    uint32 lockUntil;
    if (_value > _totalSupply) {
        lockUntil = uint32(block.timestamp + _TWO_DAYS);
    } else {
        lockUntil = uint32(block.timestamp + ((_value * _TWO_DAYS) /
_totalSupply));
    }

    if (lockUntil > _locked[_to]) {
        _locked[_to] = lockUntil;
    }
```

Then before the lock expires, the LP token cannot be transferred or burned because of the check in _beforeTokenTransfer in LP token

```
if (_locked[from] > timestamp) {
    revert GoatErrors.LiquidityLocked();
}
```

Then we can consider the griefing factor below:

1. user A mint LP token, the LP is supposed to lock for 2 days.
2. After user A lock for 2 days, user wants to burn his LP and remove liquidity in exchange for underlying token paired with WETH.
3. user B mint for LP token for user A, extending user A's lock
4. user A failed to burn his LP token.

Recommendation

Account A should not be able to interfere with Account B's lock time, mint LP token only to msg.sender that add liquidity

Protocol Response:

Acknowledged and known issue.

M1 - Loss of gas yield, WETH rebasing yield and blast point if the protocol wants to deploy the contract on blast

Vulnerability Detail

The protocol intends to deploy the contract in as many EVM network as possible.

And protocol explicitly says they want to deploy the contract in blast as well.

Blast chain is a L2 EVM optimism fork but with a few new feature:

1. the gas spent on smart contract can be claimed by smart contaract
2. Native WETH is positively rebasing and accure yield.
3. blast point is distributed to the contract point admin for future airdrop.

But because the goat trading protocol does not integrate with gas claiming and does not handle the WETH rebasing and does not config point admin.

The protocol the incur loss overtime.

Recommendation

Integration guide for claiming WETH yield:

https://docs.blast.io/building/guides/weth-yield

```
contract MyContract {
  // NOTE: these addresses differ on the Blast mainnet and testnet; the
lines below are the mainnet addresses
  IERC20Rebasing public constant USDB =
IERC20Rebasing(0x4300000000000000000000000000000000000003);
  IERC20Rebasing public constant WETH =
IERC20Rebasing(0x4300000000000000000000000000000000000004);
  // NOTE: the commented lines below are the testnet addresses
  // IERC20Rebasing public constant USDB =
IERC20Rebasing(0x4200000000000000000000000000000000000022);
  // IERC20Rebasing public constant WETH =
IERC20Rebasing(0x4200000000000000000000000000000000000023);

  address public owner;

  constructor() {
    USDB.configure(YieldMode.CLAIMABLE) //configure claimable yield for
USDB
    WETH.configure(YieldMode.CLAIMABLE) //configure claimable yield for
WETH
    owner = msg.sender;
  }

  function claimYield(address to, uint256 amount) {
    require(owner == msg.sender, "invalid owner");
    WETH.claim(to, amout);
  }
}
```

Integration guide to claim gas:

https://docs.blast.io/building/guides/gas-fees#setting-gas-mode-to-claimable

```
interface IBlast {
  // Note: the full interface for IBlast can be found below
  function configureClaimableGas() external;
  function claimAllGas(address contractAddress, address recipient) external
returns (uint256);
}

contract MyContract {
  IBlast public constant BLAST =
IBlast(0x4300000000000000000000000000000000000002);

  constructor() {
    // This sets the Gas Mode for MyContract to claimable
    BLAST.configureClaimableGas();
  }

  // Note: in production, you would likely want to restrict access to this
  function claimMyContractsGas() external {
    BLAST.claimAllGas(address(this), msg.sender);
  }
}
```

Integration guide to claim the points:

https://blastpublic.notion.site/PUBLIC-Blast-Mainnet-Points-API-f8abea9d6e67417890d4a300ecbe5827

the mainnet point address:

https://docs.blast.io/airdrop/api#mainnet-environment

Protocol Response:

Will Fix.

M2 – Take over revert if the original liquidity provider is blocklisted

Vulnerability Detail

When a user wants take over the pool with good intention to modify the virtual token balance setting, he has to pay 10% extra token

the original token is refunded to original liquidity provider:

```
    // transfer excess token to the initial liquidity provider
    IERC20(_token).safeTransfer(
        initialLpInfo.liquidityProvider, (localVars.tokenAmountForAmmOld +
    localVars.tokenAmountForPresaleOld)
    );
```

However, certain token such as USDC, USDT has blocklist feature.

https://github.com/d-xo/weird-erc20?tab=readme-ov-file#tokens-with-blocklists

If a blocklisted address create a pair with clearly invalid virtual token / virtual WETH parameter,

take over feature cease to work because transferring token to a blocklisted recipient revert the transaction.

Recommendation

Transfer the fund to a seperate escrow contract and let original liquidityProvider claim the fund to not let blocklisted recipient
block the take over transaction.

Protocol Response:

Acknowledged.

M3 – Both protocol and LP loss 40% of the fee in presale phrase

Vulnerability Detail

The logic to handle the fee is in the implementation below

```
// here either amountWethIn or amountWethOut will be zero

// fees collected will be 99 bps of the weth amount
if (amountWethIn != 0) {
    feesCollected = (amountWethIn * 99) / 10000;
} else {
    feesCollected = (amountWethOut * 10000) / 9901 - amountWethOut;
}
// lp fess is fixed 40% of the fees collected of total 99 bps
feesLp = (feesCollected * 40) / 100;

uint256 pendingProtocolFees = _pendingProtocolFees;

// lp fees only updated if it's not a presale
if (!isPresale) {
    _pendingLiquidityFees += uint112(feesLp);
    // update fees per token stored
    feesPerTokenStored += uint184((feesLp * 1e18) / totalSupply());
```

```
    }

    pendingProtocolFees += feesCollected - feesLp;
```

lp fee is fixed at 40%,

but if the pool is in presale phrase => presale phrase is True and !isPresale is False

the feesLp is not count as _pendingLiquidityFees

if the pool is in presale phrase, the feesLP is still deducted from the feesCollected

```
    pendingProtocolFees += feesCollected - feesLp;
```

This means when the pool is in presale phrase, the Lp's fee neither goes to the protocol nor goes to the LP provider, result in loss of this 40% fee.

Recommendation

Distribute the 40% Lp fee to protocol during the presale phrase.

```
    pendingProtocolFees += feesCollected - feesLp;
    if(isPresale) {
        pendingProtocolFees += feesLp;
    }
```

Protocol Response:

Acknowledged. Fees in bootstrapping are designed to go to the pool rather than the LP so this is working as intended

M4 – Withdrawal limit can be reset if the LP is minted to liquidity provider

Vulnerability Detail

When the liquidity is first added, the code set the initial lp info

```
// @note can this be an attack area to grief initial lp by using to as
initial lp?
if (mintVars.isFirstMint || to == _initialLPInfo.liquidityProvider) {
    _updateInitialLpInfo(liquidity, balanceEth, to, false, false);
}
```

Then we set the withdrawal left state to 4

```
info.fractionalBalance = uint112(((info.fractionalBalance *
info.withdrawalLeft) + liquidity) / 4);
info.withdrawalLeft = 4;
info.liquidityProvider = lp;
if (wethAmt != 0) {
    info.initialWethAdded = uint104(wethAmt);
}
```

Then every time when the new liquidity provider burn their LP in batch, the ithdrawalLeft is decremented by 1

```
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;
    }
}
```

Then after all info.withdrawalLeft is used up, the liquidity provider should not be able to burn more LP token

but this withdrawal left check can be bypassed by simply minting more LP to the liquidity provider.

If the Lp is minted to the liquidity provider again:

```
// @note can this be an attack area to grief initial lp by using to as
initial lp?
if (mintVars.isFirstMint || to == _initialLPInfo.liquidityProvider) {
    _updateInitialLpInfo(liquidity, balanceEth, to, false, false);
}
```

to == _initialLPInfo.liquidityProvider is true, and _updateInitialLpInfo is called again to reset the withdrawal limit.

Recommendation

Does not allow resetting the withdrawal limit for the same liquidityProvider

```
{
info.fractionalBalance = uint112(((info.fractionalBalance *
info.withdrawalLeft) + liquidity) / 4);
if (info.liquidityProvider != lp) {
 info.withdrawalLeft = 4;
}
info.liquidityProvider = lp;
if (wethAmt != 0) {
    info.initialWethAdded = uint104(wethAmt);
}
```

Protocol Response:

Acknowledged. Withdrawal limits are only for initial liquidity.

M5 – Flashloan can steal LP Fees

Vulnerability Detail

When user wants to withdraw the fees,

the needs to call the function withdrawFees

```
    function withdrawFees(address to) external {
        uint256 totalFees = _earned(to, feesPerTokenStored);

        if (totalFees != 0) {
            feesPerTokenPaid[to] = feesPerTokenStored;
            lpFees[to] = 0;
            _pendingLiquidityFees -= uint112(totalFees);
            IERC20(_weth).safeTransfer(to, totalFees);
        }
        // is there a need to check if weth balance is in sync with reserve
    and fees?
    }
```

As we can see, the earned is fee is computed by calling _earned

```
    function _earned(address lp, uint256 _feesPerTokenStored) internal view
    returns (uint256) {
        uint256 feesPerToken = _feesPerTokenStored - feesPerTokenPaid[lp];
        uint256 feesAccrued = (balanceOf(lp) * feesPerToken) / 1e18;
        return lpFees[lp] + feesAccrued;
    }
```

the feesAccrued comes from

```
    uint256 feesAccrued = (balanceOf(lp) * feesPerToken) / 1e18;
```

Then a single user can

1. flash loan the token
2. paired with WETH to add liquidity to mint a large amount of LP token
3. withdraw the fee because inflating balanceOf(lp) inflating the feesAccrued

While there is a lock mechanism to make sure the LP cannot be burnt immediately

If the accured fee amount is high, the flashloan still steal other user's LP fee.

Recommendation

Consider adding the check to make sure if user mint LP, the LP does not accures fees immediately within same transaction.

Protocol Response:

An update to earned is made on minting and burning so the protocol claim that this is not a valid finding.

M6 - New liquidity provider can bypass the 7 day withdrawal timelock for one time.

Vulnerability Detail

The function call in Pair contract is used to update the updateInitialLpInfo information.

```
_updateInitialLpInfo(liquidity, balanceEth, to, false, false);
```

this is calling

```
 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 {
            info.fractionalBalance = uint112(((info.fractionalBalance *
info.withdrawalLeft) + liquidity) / 4);
            info.withdrawalLeft = 4;
            info.liquidityProvider = lp;
            if (wethAmt != 0) {
                info.initialWethAdded = uint104(wethAmt);
            }
        }

        // Update initial liquidity provider info
        _initialLPInfo = info;
    }
```

If the isBurn and internalBurn are both set to false, the code enter the block

```
info.fractionalBalance = uint112(((info.fractionalBalance *
info.withdrawalLeft) + liquidity) / 4);
info.withdrawalLeft = 4;
info.liquidityProvider = lp;
if (wethAmt != 0) {
    info.initialWethAdded = uint104(wethAmt);
}
```

When the pool is taken over, the function

```
_updateInitialLpInfo(liquidity, balanceEth, to, false, false);
```

the function above would update the new liquidity provider address,

but the info.lastWithdraw is not reset, and the info.lastWithdraw still comes from the old liquidity provider.

then suppose at day 10, the old liquidity provider burn their LP, at day 18 days, the pool is taken over by new liquidity provider,

the new liquidity provider earns free one time chance to burning LP and by bypass the check below

```
function _beforeTokenTransfer(address from, address to, uint256 amount)
internal override {
    GoatTypes.InitialLPInfo memory lpInfo = _initialLPInfo;
    if (to == lpInfo.liquidityProvider) revert
GoatErrors.TransferToInitialLpRestricted();
    uint256 timestamp = block.timestamp;
    if (from == lpInfo.liquidityProvider) {
        // initial lp can't transfer funds to other addresses
        if (to != address(this)) revert
GoatErrors.TransferFromInitialLpRestricted();

        // check for coldown period
        if ((timestamp - 1 weeks) < lpInfo.lastWithdraw) {
            revert GoatErrors.WithdrawalCooldownActive();
        }
```

Recommendation

set the info.lastWithdraw tp current block.timestamp when new lp liquidity provider address is set.

```
 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 {
    info.fractionalBalance = uint112(((info.fractionalBalance *
info.withdrawalLeft) + liquidity) / 4);
    info.withdrawalLeft = 4;
    info.liquidityProvider = lp;
    info.lastWithdraw = uint32(block.timestamp);
    if (wethAmt != 0) {
        info.initialWethAdded = uint104(wethAmt);
    }
}
```

Protocol Response:

Takeover can only happen during presale and liquidity can’t be withdrawn during presale.

Balance check is too strict and transation can be griefed

Vulnerability Detail

If there is one withdrawalLeft and the liquidity provider wants to burn all his lp

```
    // we only check for fractional balance if withdrawalLeft is not 1
    // because last withdraw should be allowed to remove the dust amount
    // as well that's not in the fractional balance that's caused due
    // to division by 4
    if (lpInfo.withdrawalLeft == 1) {
        uint256 remainingLpBalance = balanceOf(lpInfo.liquidityProvider);
        if (amount != remainingLpBalance) {
            revert GoatErrors.ShouldWithdrawAllBalance();
        }
    }
```

the liquidity provider input amount as 10000 amount of token,

user A can frontrun the liquidity provider's transaction and transfer 1 wei of LP token to liquidity provider address,

then transaction will revert because amount != remainingLpBalance

```
    if (amount != remainingLpBalance) {
        revert GoatErrors.ShouldWithdrawAllBalance();
    }
```

Same too strict balance check is in the function mint as well:

```
    uint256 balanceToken = IERC20(_token).balanceOf(address(this));

    if (balanceEth < mintVars.bootstrapEth) {
        (uint256 tokenAmtForPresale, uint256 tokenAmtForAmm) =
    _tokenAmountsForLiquidityBootstrap(
            mintVars.virtualEth, mintVars.bootstrapEth, balanceEth,
    mintVars.initialTokenMatch
        );
        if (balanceToken != (tokenAmtForPresale + tokenAmtForAmm)) {
            revert GoatErrors.InsufficientTokenAmount();
        }
        liquidity =
            Math.sqrt(uint256(mintVars.virtualEth) *
    uint256(mintVars.initialTokenMatch)) - MINIMUM_LIQUIDITY;
    }
```

if another user transfer 1 wei of token before when balanceEth < mintVars.bootstrapEth

transaction will revert because balanceToken != (tokenAmtForPresale + tokenAmtForAmm

```
    if (balanceToken != (tokenAmtForPresale + tokenAmtForAmm)) {
        revert GoatErrors.InsufficientTokenAmount();
    }
```

Recommendation

Avoid too strict balance check, do not use != or ==, can use >=

Protocol Response:

Will Fix.

L1 – User should not able to add liquidity to a removed pair

Vulnerability Detail

After 30 days if the trading pool failed to raise enough liquidtiy,

the liquidity provider can call GoatV1Pair.sol#withdrawExcessToken

but after the pair is removed,

user can still mint (add liquidity) to the removed pool / burn (remove liquidity to the removed pool)

At this point, The liquidity provider can call GoatV1Pair.sol#withdrawExcessToken again to convert the pool to AMM while withdraw the token again

```
    IERC20 token = IERC20(_token);
    uint256 poolTokenBalance = token.balanceOf(address(this));
    uint256 amountToTransferBack = poolTokenBalance - tokenAmtForAmm;
```

Recommendation

Does not allow user to mint (add liquidity) after a pair is removed

```
    token.safeTransfer(initialLiquidityProvider, amountToTransferBack);

    if (reserveEth != 0) {
        _burnLiquidityAndConvertToAmm(reserveEth, tokenAmtForAmm);
        // update bootstrap eth because original bootstrap eth was not met and
        // eth we raised until this point should be considered as bootstrap eth
        _bootstrapEth = uint112(bootstrapEth);
        _update(reserveEth, tokenAmtForAmm, true);
    } else {
        pairRemoved = True;
        IGoatV1Factory(factory).removePair(_token);
    }
```

and in the mint function, first check

```
    if (pairRemoved) {
        revert("pool is removed")
    }
```

Protocol Response:

Will Fix.

L2 – MEV protection can user transaction with no sandwich intention

Vulnerability Detail

the protocol has built-in sandwich protection (MEV protection)

```
    function _handleMevCheck(bool isBuy) internal returns (uint32 lastTrade) {
        // @note  Known bug for chains that have block time less than 2 second
        uint8 swapType = isBuy ? 1 : 2;
        uint32 timestamp = uint32(block.timestamp);
```

```
        lastTrade = _lastTrade;
        if (lastTrade < timestamp) {
            lastTrade = timestamp;
        } else if (lastTrade == timestamp) {
            lastTrade = timestamp + swapType;
        } else if (lastTrade == timestamp + 1) {
            if (swapType == 2) {
                revert GoatErrors.MevDetected1();
            }
        } else if (lastTrade == timestamp + 2) {
            if (swapType == 1) {
                revert GoatErrors.MevDetected2();
            }
        } else {
            // make it bullet proof
            revert GoatErrors.MevDetected();
        }
        // update last trade
        _lastTrade = lastTrade;
    }
```

As protocol also mentioned in the documentation:

https://goattrading.gitbook.io/goat/features/for-traders/sandwich-bot-protection

> Previously the protection was simply that no more than 2 transactions could happen in a single pool. This worked perfectly to avoid sandwich bots since they were no longer able to sandwich free from risk, but if many transactions were occurring at once, such as one a token first launches, there were many failing purchases.

> The advancements we've made with Goat are that the sandwich protection focuses on preventing a "switch" of trading directions rather than denying based on a simple number of trades. For example, if 50 transactions that are all buys occur in a single block, they will all be let through since no sandwich can be occurring. If there's 1 sell then 49 buys after, that will also be allowed since there's only 1 switch of trading direction. However, if there's 1 sell then 1 buy then another sell, that's 2 switches of trading direction and the last transaction will be reverted.

note:

> if there's 1 sell then 1 buy then another sell, that's 2 switches of trading direction and the last transaction will be reverted.

but the last sell may not comes from MEV bot that send sandwich transaction, which means user's regular sell transaction may consistently revert, after 1 sell and 1 buy.

Recommendation

I think if user is concern about slippage, the minOutput parameter should already do the job to make sure if they get sandwich, transaction revert.

Protocol Response:

This is a known issue and already highlighted in the documentation.

Bug Fix From TrungOre private audit

Fix 1:

The fix ensure that tokenAmtForAmm will not be truncated because of division before multiplication.

https://github.com/inedibleX/goat-trading/pull/2/commits/984bd93ba6a44156b6e9bb6f2f9efdaf24a589d4

Fix 2:

The fix ensure that the code uses reserveETH and reserve Token exclduing the WETH and Token amount.

https://github.com/inedibleX/goat-trading/pull/2/commits/def5c2fd3e5df7cd8623b5a1f6f32044f2014b2a

Fix 3:

The fix ensure that the burn function cannot be triggered during presale period to mitigate sandwich attack in initial LP.

https://github.com/inedibleX/goat-trading/pull/2/commits/c458b28663578c86966ec8475993769c2e1a06b1

Fix 4

The fix mitigate the arithmic underflow when adding liquidity

https://github.com/inedibleX/goat-trading/pull/2/commits/06ae7085f4a12ad87bfc2ec2b6e716d6f6c2bf06

Fix 5:

The fix validate that k invariant will not be bypassed during presale period.

https://github.com/inedibleX/goat-trading/pull/2/commits/d38e1ab5bc1eef81f022f119278e58b30b391479