SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Private
Prepared for:	Aloe
Prepared by:	Sherlock
Lead Security Expert:	panprog
Dates Audited:	July 8 - July 16, 2024
Prepared on:	September 24, 2024

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: e384e905e437a90356eb2c8db686e956f95ee327

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	0

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

panprog

Issue M-1: Uniswap governance increase of protocol fees will instantly spike the target IV down artificially deflating IV and reducing collateral requirements for borrowers below protocol expected values

Source: https://github.com/sherlock-audit/2024-06-aloe-update-judging/issues/1

The protocol has acknowledged this issue.

Found by

panprog

Summary

Aloe calculates IV based on feeGrowthGlobal (`fgg` which is the fees per liquidity unit collected over the `FEE_GROWTH_AVG_WINDOW`) and `gamma` (percentage of fees received by LPs from the trading volume, which is the trading fee percentage less protocol fee percentage), more specifically IV is in direct proportion to `sqrt(fgg)` and to `sqrt(gamma)`. If Uniswap Governance changes the protocol fees setting, `fgg` remains the same, but the `gamma` value instantly drops, meaning calculated new IV will also instantly drop (by `sqrt(gamma_new / gamma_old)`). This will cause a steady decline of IV over the next `FEE_GROWTH_AVG_WINDOW` (72 hours) regardless of market conditions, meaning the IV will be artificially deflated. Since the protocol asseses borrowers health based on IV, this deflated IV will lead to more risky borrower positions becoming acceptable with increased risk of bad debt liquidations, and bad debt will cause lenders to lose funds since they won't be able to withdraw all funds due to bad debt. This breaks protocol risk expectations.

Root Cause

The `FeeGrowthGlobals` stored in the protocol are just direct values from the uniswap pool, meaning this is just raw fees collected per liquidity unit: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src/VolatilityOracle.sol#L149-L156

If/when uniswap governance changes protocol fees, anyone can call `VolatilityOracle.prepare` to reflect these changes in the internal cache: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src/VolatilityOracle.sol#L39-L46

Once `prepare` is called, `gamma` values instantly change to a new value. And `gamma` value is used directly in `IV` calculations: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src/libraries/Volatility.sol#L61-L85

This causes an instant change in `IV` value as well. So the root cause is that only collected fees (`fgg`) are stored during VolatilityOralce updates instead of `volume` amount derived from it (`fgg * gamma`).

Internal pre-conditions

The only condition is for any user to call `VolatilityOracle.prepare` to reflect uniswap protocol fees in the internal cache. Since this is permissionless function, it's very easy for anyone to call it.

External pre-conditions

Uniswap governance increases protocol fees.

Attack Path

Once pre-conditions are met, nothing is needed to do - IV will always drop regardless of any other actions.

Impact

New (target) IV instantly drops. Due to per-second and per-update IV change limitations, the actual IV used in collateral calculations will predictably go down slowly. So while this limits the impact, it still causes IV to decline below expected levels temporarily. In the worst case, the protocol fees are set from 0 to `1/4` (max value allowed in uniswap). This will cause the target IV to decrease by ~14%. The actual IV decrease will be limited by IV change safeguards and will depend on circumstances. Since per-update (every 4 hours) IV only actually changes by 1% of difference between target and current volatility, this means that over the 72 hours window the IV will decrease by at most 2.52%. This might be more impactful if actual volatility increases significantly during the same time, but the issue described here will not allow IV to increase and will keep it decreasing.

The incorrect (deflated) IV will cause more risky borrower positions to be healthy which increases risk of bad debt, which is a loss for the lenders and possible bank run, since the last lenders to withdraw will be unable to do so due to bad debt.

PoC

Not needed (instant change/jump of target IV after uniswap protocol fee change is obvious from the code).

Mitigation

`VolatilityOracle` should store "volume per liquidity" cumulative values (which do not depend on any protocol parameter) instead of cumulative collected fees. Since IV calculations use `fgg`s and `gamma`s in the following formula: `fgg0 * gamma1 * sqrt(P) + fgg1 * gamma0 / sqrt(P)`, cumulative values of `fgg0 * gamma1` and `fgg1 * gamma0` might be stored, maybe even along with `sqrt(P)` terms as well, which might also increase precision of calculations. Creating such cumulative value(s) should be pretty straighforward.

Discussion

haydenshively

Good callout and interesting proposed mitigation, but I think this should be Low. A change of 2-3% over the course of 72 hours is well within expectations for the protocol. Also, changes to the Uniswap protocol fee would garner lots of attention in the ecosystem, and I think it's reasonable to expect borrowers to be aware/prepared for the IV change.

Issue M-2: Uniswap governance increase of protocol fees will break IV calculations making it permanently deflated compared to actual volatility.

Source: https://github.com/sherlock-audit/2024-06-aloe-update-judging/issues/2

The protocol has acknowledged this issue.

Found by

panprog

Summary

There is a conceptual issue with protocol math, which will permanently reduce IV if uniswap governance increases protocol fees. This, in turn, will significantly increase a risk of bad debt liquidations since IV is used for borrowers health calculations and borrowers will be allowed more risky positions than designed by the protocol.

Aloe uses the following formula to calculate IV: `IV = 2 * sqrt( 20000 * (fgg0 · gamma1 · sqrt(P) + fgg1 · gamma0 / sqrt(P)) / tickSpacing )`

where `fgg` is the fees per liquidity unit collected over the `FEE_GROWTH_AVG_WINDOW` (72 hours) and `gamma` is the percentage of fees received by LPs from the trading volume (`gamma = trading fee - protocol fee`). As seen from the formula, IV is in direct proportion to `sqrt(fgg)` and to `sqrt(gamma)`.

Increase of protocol fees decreases gamma. In such case in efficient market `fgg` must stay about the same (since `fgg` is basically APR% earned by LPs, so if they start earning less than before due to increased protocol fees / decreased gamma, it's likely some liquidity will withdraw for better opportunities elsewhere which will bring per-liquidity earnings (`fgg`) back to about the same value). In reality things are more complex (reduced liquidity will cause reduced trading volume and reduced fees, which in turn can cause more liquidity to withdraw) are `fgg` can slightly decrease, but in these circumstances it can never increase.

All in all, if uniswap governance increases protocol fees, this causes `gamma` to decrease and `fgg` to either stay the same or slightly decrease, meaning `IV` will **permanently** decrease (even though actual volatility doesn't change).

Root Cause

`IV` is proportional to `sqrt(gamma)` and to `sqrt(fgg)`: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src/libraries/Volatility.sol#L61-L85

If protocol fees are increased, `gamma` permanently decreases and `fgg` stays the same or decreases (but never increases), leading to permanent IV decrease.

The root cause is conceptual issue with the IV calculations math. In my understanding it must not depend on `gamma`, only on `fgg`, because intuitively IV shouldn't depend on how much fee percentage is taken from trading, as change of fee will just adjust liquidity to keep `fgg` about the same. But this requires further research.

Internal pre-conditions

The only condition is for any user to call `VolatilityOracle.prepare` to reflect uniswap protocol fees in the internal cache. Since this is permissionless function, it's very easy for anyone to call it.

External pre-conditions

Uniswap governance increases protocol fees.

Attack Path

Once pre-conditions are met, nothing is needed to do - IV will always permanently decrease compared to actual volatility.

Impact

In worst case, if protocol fees are set to its max value of `1/4`, new `gamma` is then `3/4` of old `gamma`, and new IV will be `sqrt(3/4) = 87%` of the old IV **permanently**. So the impact is permanent reduction of IV by 13%. Per-second and per-update IV change limitations are irrelevant, because this is the long-term IV so it will slowly come down to the new (reduced) amount and will stay there.

This deflated IV will allow borrowers to open more risky positions than they should be allowed by protocol, thus the risk of bad debt significantly increases, and in case of bad debt, lenders will lose funds as not all lenders will be able to withdraw due to bad debt, and this can cause bank run since the last to withdraw will be unable to do so.

Additionally, in previous contest the team indicated that IV calculation is very important to the protocol, so IV calculations should be correct, and this issue demonstrates a permanent error in IV calculation by 13%.

PoC

Not needed as this is conceptual math issue.

Mitigation

Since the IV calculations is based on experimental formula, possibly conduct a deeper research to see if it's possible to get rid of `gamma` in the IV calculations, as in my understanding IV must not depend on `gamma`, only on fees per liquidity, which should be enough.

Discussion

haydenshively

Eqn 6.9 of the Uniswap v3 Whitepaper states that

$$\Delta f_{g,1} = y_{in} \cdot \gamma \cdot (1 - \phi)$$

which, by footnote 6, is equivalent to

$$\Delta f_{g,1} = y_{in} \cdot \left(\gamma - \frac{\gamma}{\text{protocolFee}}\right)$$

Therefore, as you point out in #1, the instantaneous effect of changing the `protocolFee` is to change the measured IV by a factor of $\sqrt{\frac{1-\phi_{old}}{1-\phi_{new}}}$. Since ϕ is between 0 and $1/4$, the maximum change is +15% (but of course this *measured* value is subjected to the EMA and max change constraints before being used in the protocol).

Let's look at the scenario where ϕ is changed from 0 to $1/4$:

- initially, IV increases by around 15% due to the math above
- since LPs are now receiving less fees for the same volume, they start to withdraw liquidity
- less liquidity means higher slippage for traders, so they swap less
- these two factors lead to less activity in the pool -- lower IV
- **Assuming the previous IV was "correct", the effects should cancel out. The instantaneous 15% increase should be followed by a gradual 15% decrease as the market reaches a new equilibrium.**

panprog

@haydenshively Where does the math imply increase in IV if protocol fee is increased? It's the opposite, IV will decrease.

Protocol fees = 0:

$$\Delta f_{g,1} = 10000 \cdot (0.0003) = 3$$

Protocol fees = 1/4:

$$\Delta f_{g,1} = 10000 \cdot \left(0.0003 - \frac{0.0003}{4}\right) = 2.25$$

If uniswap protocol fees are changed from 0 to 1/4, the change in measured IV is **-15%** (not **+15%**), so:

- initially, IV **decreases** by around 15% due to the math above
- since LPs are now receiving less fees for the same volume, they start to withdraw liquidity
- less liquidity means higher slippage for traders, so they swap less
- these two factors lead to less activity in the pool -- *even lower* IV
- Assuming the previous IV was "correct", the instantaneous 15% decrease should be followed by a further gradual decrease as the market reaches a new equilibrium (or it can stay at ~15% decrease if fees/liquidity ratio returns back to previous level)

To add on: actually the way it is now, target IV will instantly drop by 13% (due to instant gamma drop), and then if nothing else changes (no liquidity withdrawn), it will gradually reduce by another 13% (due to fee growth global reduction from the math above - but it will fully reflect in target IV only after the 72 hours since mean value of fgg is used). However, as we can reasonably expect some liquidity to withdraw, the gradual decline will be less than 13% (0% in best case). 13% of gamma drop will still reduce overall IV by at least 13% from the original "correct" IV.

haydenshively

Apologies, you are correct about $\Delta f_{g,1}$ dropping.

What I meant to say is that our estimate of volume is still correct. As specified on this line, we divide by the cached value of gamma. Essentially:

$$\gamma_{cached} = \gamma - \frac{\gamma}{\text{protocolFee}}$$

$$\Delta f_{g,1} = y_{in} \cdot \left(\gamma - \frac{\gamma}{\text{protocolFee}}\right) = y_{in} \cdot \gamma_{cached}$$

$$y_{in} = \frac{\Delta f_{g,1}}{\gamma_{cached}}$$

All that to say, our measurement of volume is not impacted by a change to the `protocolFee`. Neither is our measurement of liquidity. This means we can restrict our analysis to the leading coefficient in the IV equation: $\sqrt{\gamma_0 \cdot \gamma_1}$

All else being equal, we get:

$$\frac{IV_{new}}{IV_{old}} \propto \frac{\sqrt{\gamma_{0,new} \cdot \gamma_{1,new}}}{\sqrt{\gamma_{0,old} \cdot \gamma_{1,old}}} = \sqrt{\left(1 - \frac{1}{protocolFee_0}\right) \cdot \left(1 - \frac{1}{protocolFee_1}\right)}$$

So if one of protocol fee is turned up to 4, we do in fact get a drop of 15%. If *both* are turned up to 4 at the same time, we get an instantaneous drop of 25% (!!)

That said, I'm still not worried about this for the following reasons:

- The formula is still "correct" based on this, and as such the before/after IV's should match up in an efficient market, even if we're failing to reason about it here. In other words, we must be wrong about the extent to which volume and liquidity decrease (the ratio very much matters, and it's hard to predict a priori)
- If IV does get permanently stuck lower, governance could increase `nSigma` to compensate.
- The instantaneous increase in *measured* IV is tempered by the maximum per-update rate of change and the EMA.

panprog

@haydenshively What I describe in this issue is that the article you base your formulas on doesn't account for the protocol fee and I show *why* new IV after protocol fee increase will not match up the old IV in efficient market. Yes, the ratio of volume and liquidity does matter, however new IV can match old IV if and only if new `volume/liquidity` increases by at least 25%. Any other ratio of new `volume/liquidity` (small increase, the same or lower) will decrease the new IV. And it's obvious that when the trading fee is the same, but liquidity earnings are *decreased*, this ratio can not increase: if `volume/liquidity` increases, this means that traders start trading **more** when liquidity is the same and trading fees are the same => this is hightly unlikely and protocol fees increase **can not** be the reason for such behaviour in efficient market.

Basically in efficient market after protocol fees increase:

- Traders require `volume/liquidity` to be the same as trading fee is unchanged
- Liquidity providers require `volume_new/liquidity_new = 1.25 * volume_old/liquidity_old` because the fee they receive has decreased by 25%

As liquidity providers requirement is not satisfied, they will withdraw liquidity. This will cause traders requirement to stop being satisfied, thus traders will reduce volume and so on. In really efficient market this will probably cause both liquidity and trading volume to go to 0, because at no point both traders and liquidity providers requirements are satisfied. However, in real environment it's likely that each trader and liquidity provider has different tolerance to ROI decrease, so at some point this process will stop, and most likely this will be some mid-point between traders requirements and liquidity providers requirements, so something around `1.13 * volume_old / liquidity_old`, meaning new permanent calculated IV will be about 13% lower than old IV. It's highly unlikely that the stop point will be `1.25 * volume_old / liquidity_old`, meaning it's highly unlikely that new IV = old IV in efficient market.

haydenshively

Acknowledged. Valid medium/low but won't fix.

We can (will) adjust this in future versions of the protocol, but I think it's ok for now, based on the reasons I gave here and the fact that Uniswap governance has thus far been unwilling to actually enable the protocol fee.

Issue M-3: Long time delay between transactions causes RESERVE account to lose rewards

Source: https://github.com/sherlock-audit/2024-06-aloe-update-judging/issues/3

The protocol has acknowledged this issue.

Found by

panprog

Summary

Protocol Q&A mentions the RESERVE address:

> There is a RESERVE address to which a portion of all interest accrues. It should be able to take any action and behave normally.

However, when RESERVE address receives its portion of all interest, it also receives rewards, but the rewards are calculated incorrectly, adding less rewards to RESERVE than it should receive, thus the RESERVE account loses rewards. This happens because RESERVE receives its portion of all interest continuously (each second), but the rewards are calculated with RESERVE balance only at the start of the interval (RESERVE balance at the time of last transaction) ignoring any RESERVE balance accural inside the interval.

Example: transaction at t=0: RESERVE balance = 0, reward rate = 1 per second per unit balance, total borrows increases at such a rate, the RESERVE receives 10 new tokens per second t=1: RESERVE balance = 10, rewards should be 0 (not updated in storage) t=2: RESERVE balance = 20, rewards should be 10 (not updated in storage) t=3: RESERVE balance = 30, rewards should be 10+20=30 (not updated in storage) t=4: RESERVE balance = 40, rewards should be 30+30=60 (not updated in storage) transaction at t=5: first the rewards are updated for the time interval [0;5] with the **stored** balance (0), so RESERVE rewards are 0, only then the RESERVE balance is set to 50.

Root Cause

RESERVE receives its portion of all interest by minting new shares: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src/Lender.sol#L527

However, when minting, the rewards are calculated by multiplying user's balance before the mint by the accumulated rewards per unit balance: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src

/Lender.sol#L435-L437 https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src/libraries/Rewards.sol#L92

This is correct for all accounts **except** RESERVE: all accounts have static balances between updates, so the rewards formula is correct, but RESERVE has continously increasing balance over time, meaning the formula to calculate the reward should take this into account, but it doesn't.

Internal pre-conditions

Always happens by itself, the longer the time between transactions, the larger the rewards loss for the RESERVE account.

External pre-conditions

None

Attack Path

None, always happens by itself.

Impact

RESERVE account always losing rewards, the longer the time between transactions, the larger the rewards loss. For example, if the protocol has $10M borrowed at the rate of APR of 10%, and reserve factor is 10%, this means that RESERVE account receives $0.003 per second. Let's say there is a reward at the rate of 1 reward token per unit balance per second. Then:

- if transactions are going every second for 1 hour, the RESERVE account receives reward of 0.003 * (0 + 1 + 2 + 3 + 4 + ... + 3599) = 0.003 * 6478200 = 19434.6 tokens (this is the correct amount it should receive)
- if there are just 2 transactions (at t = 0 and at t = 1 hour), then RESERVE account receives reward of 0. If both scenarios continue for 1 day (transactions every second vs every 1 hour), then in scenario 1 the RESERVE account will receive 11.197M tokens, in scenario 2 the RESERVE account will receive 10.731M tokens, a loss of 466K reward tokens or a 4.3% loss.

The RESERVE account total rewards accured in time is quadratic, while the rewards loss is linear in time, so the longer time period, the higher the absolute loss and the smaller the relative (percentage) loss.

PoC

N/A

Mitigation

When minting portion of percentage to RESERVE account, handle the rewards accumulation in a different way - use the simple arithmetic progression formula both when accumulating global rewards (linearly increasing totalSupply) and RESERVE user's rewards (linearly increasing balance). In reality the totalSupply and balance growth is exponential (but very close to linear in small intervals), but this might not be worth the effort since the error between linear and exponential growth calculations will be very tiny.

Discussion

haydenshively

> RESERVE has continously increasing balance over time, meaning the formula to calculate the reward should take this into account, but it doesn't

We do *think* of the `RESERVE` as having a continuously increasing balance, but technically it doesn't. Its balance doesn't actually increase until the pool is touched / interest is accrued. So depending on how you look at it, you could argue that this style of rewards accrual is actually correct.

Either way, I think this would be Low, but we'll definitely make a note of it in our docs.

panprog

If we consider 2 different scenarios:

1. Protocol transactions come at 1 per second => RESERVE receives 11.197M tokens reward
2. Protocol transactions come at 1 per hour => RESERVE receives 10.731M tokens reward

This is a definite loss of 4% for RESERVE account only because transactions are more rare. Or maybe said in different way, the RESERVE account must do empty transactions very frequently in order not to lose funds. The technical update of balance during transactions is simply because it's technically impossible to update anything in blockchain without transaction, so I don't think this is a valid argument. The formulas obviously imply continuous increase of balance.

haydenshively

The underlying value of *already-owned-shares* does continuously increase; the argument I'm making is that `RESERVE` does NOT already own the shares, and thus is not owed the additional rewards.

I'll admit it's strange for the frequency of txn in the pool to impact the `RESERVE`'s rewards rate, but I don't think it's a bug. If anything, this is a design decision favoring non-RESERVE users.

WangSecurity

@haydenshively thank you for reflecting on this issue, but we'll have to keep it as valid based on the following statement in the README:

> There is a RESERVE address to which a portion of all interest accrues. It should be able to take any action and behave normally

@panprog showed how it the RESERVE is not behaving normally and receives less rewards, hence, we have to keep it medium.

Issue M-4: Any user can DOS `BorrowNFT.modify` of any other user by front-running and sending 1 wei of ETH

Source: https://github.com/sherlock-audit/2024-06-aloe-update-judging/issues/4

Found by

panprog

Summary

ETH balance is required to be 0 at the end of `modify`:

```
require(address(this).balance == 0, "Aloe: antes sum");
```

This allows any user to front-run this transaction, deposit 1 wei of ETH (for example via calling payable `multicall` function with 1 wei and empty array), and the `modify` transaction will revert in the require referenced above.

This can be kept forever only at the gas cost, DOS'ing the modification of position, making it impossible for the other user(s) to do anything with their accounts, especially withdrawing/borrowing, which can be a lost opportunity or loss of funds for the user (for example, user wanted to withdraw collateral to fund another position to avoid liquidation, but due to DOS he can't withdraw + deposit and is liquidated).

Root Cause

`BorrowNFT.modify` sends out ETH to all user borrower accounts for ante, requiring remaining balance in the `BorrowNFT` contract to be exactly 0: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/periphery/src/borrower-nft/BorrowerNFT.sol#L111

Since user doesn't control amount of ETH in the contract (and it's easy to send ETH to the contract via payable `multicall` or `mint` functions), it's easy to DOS the `modify` function by front-running and sending 1 wei of ETH to make the final balance non-zero and revert.

Internal pre-conditions

Any user call `BorrowNFT.modify`

External pre-conditions

None

Attack Path

Front-running `BorrowNFT.modify` with depositing 1 wei of ETH to `BorrowNFT` via `multicall` payable function with empty array.

Impact

DOS the NFT borrow position modification of any user. This makes it impossible for the user(s) to do any actions with their accounts, such as withdrawals, borrows etc. This can be a lost opportunity or loss of funds for the user (for example, user wanted to withdraw collateral to fund another position to avoid liquidation, but due to DOS he can't withdraw + deposit and is liquidated).

PoC

N/A

Mitigation

Consider sending remainder of ETH back to user instead of requiring it to be 0.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/aloelabs/aloe-ii/pull/242

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-5: Address approved by user in `BorrowerNFT` for one of user NFTs can steal user funds from the other NFTs even without approval, when user uses `Permit2Manager`

Source: https://github.com/sherlock-audit/2024-06-aloe-update-judging/issues/7

The protocol has acknowledged this issue.

Found by

panprog

Summary

Insufficient data verification in `Permit2Manager` allows to break security control and front-run the user using his permit to deposit funds into his NFT and deposit them into the attacker's NFT instead.

In more details: `BorrowerNFT` allows any user to mint NFT's, which creates a borrower account and allows the user to control this account via the `BorrowerNFT.modify` calls. User can either call `modify` himself, or approve any address to call it on his behalf. The user can either approve address for all of his NFTs, or he can approve address for his individual NFTs. Notice: if the user approves some address to one of his NFTs, this address must not have any control over the other NFTs of the same user. The issue described in this report breaks this condition, allowing the address approved by user to one NFT to steal user funds from the other NFTs (even when not approved for them).

Suppose attacker address is approved by user to `NFT1`. And then the same user calls `BorrowerNFT.modify` for `NFT2` with `Permit2Manager` in the calldata. Attacker can extract the user's permit to deposit funds into `NFT2` and front-run the transaction with his own transaction calling `BorrowerNFT.modify` for `NFT1` with the same owner (user), with `Permit2Manager` and permit from user's transaction (but setting the `to` address to `NFT1`).

Here are all the security control checks, all of them will be passed by the attacker's transaction:

1. `modify` first verifies if attacker is authorized, he is not, so `authorized` is set to `false`:

```
bool authorized = msg.sender == owner || isApprovedForAll[owner][msg.sender];
```

2. `modify` checks if attacker is approved to control the `NFT1`, he was approved, so this require passes correctly:

```
if (!authorized) require(msg.sender == getApproved[tokenId], "NOT_AUTHORIZED");
```

3. Next `modify` calls `borrower.modify` (where borrower corresponds to `NFT1`) with `Permit2Manager.callback`, which performs its own check. First check passes (`msg.sender` is `NFT1` borrower which is registered as borrower in the `FACTORY` and the `owner` of this borrower is `BORROWER_NFT`):

```
require(FACTORY.isBorrower(msg.sender) && owner == BORROWER_NFT, "Aloe: bad
↪   caller");
```

4. Next check passes: (a) correct permit selector, (b) `to` field corresponds to NFT1 borrower and matches `msg.sender` - this field is modified by attacker from the original user's transaction, but it passes, because signature of `permit` doesn't include `to`, so any `to` field will be accepted by `Permit2`, (c) signer is owner, which is the same both for NFT1 and NFT2, allowing attacker to reuse his signature but for a different NFT.

```
// Before calling `PERMIT2`, verify
// (a) correct function selector
// (b) `to` field is the Borrower (`msg.sender`)
// (c) correct signer address, i.e. [claimed Permit2 signer] == [user who owns
↪   the Borrower]
// Note that data[:20] is the true owner prepended by the `BORROWER_NFT`
require(
    bytes4(dataPermit2[:4]) == IPermit2.permitTransferFrom.selector &&
        bytes20(dataPermit2[144:164]) == bytes20(msg.sender) &&
        bytes20(dataPermit2[208:228]) == bytes20(data[:20])
);
```

5. Finally, the `permit2` call transfers funds with permit from unsuspecting user to `NFT1` (as described above, the `to` field is not part of the signature, so the attacker can reuse the same permit to transfer funds to a different NFT).

```
// Make calls
bool success;
(success, ) = address(PERMIT2).call(dataPermit2); // solhint-disable-line
↪   avoid-low-level-calls
if (!success) revert Permit2CallFailed();
```

6. Since attacker has full control over `NFT1`, he can immediately withdraw the funds he just stole.

Root Cause

`Permit2Manager` doesn't verify if the original `modify` caller has access to the NFT `permit` is supposed to transfer funds to, all checks are indirect.
https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/periphery/src/managers/Permit2Manager.sol#L33-L73

Lack of direct check for NFT `permit` was given to allows to modify part of the permit which doesn't require signature and still pass all indirect checks, allowing to use `permit` with **any** NFT of the same owner.

Internal pre-conditions

User has several `BorrowerNFT`s and approves attacker for only one of the NFTs, then uses `Permit2Manager` for any of the other NFTs.

External pre-conditions

None

Attack Path

1. Attacker listens to the user transaction, takes `permit` data from it, changes `to` address to NFT he is approved to and front-runs the transaction with the same transaction of his own but for the NFT he's approved for. This allows to take funds from the user with this permit and deposit them into the NFT attacker controls.
2. Attacker immediately withdraws these funds from the NFT

Impact

Attacker can break `BorrowerNFT` approvals and take funds belonging to NFT the attacker is not approved for.

PoC

N/A

Mitigation

Consider to require the owner to sign full permit data for `Permit2Manager` (including `to` field) in addition to permit itself.

Issue M-6: Address approved by user in `BorrowerNFT` for one of user NFTs can steal user Uniswap NFT when user uses `UniswapNFTManager` or `BoostManager` for the other NFTs

Source: https://github.com/sherlock-audit/2024-06-aloe-update-judging/issues/8

The protocol has acknowledged this issue.

Found by

panprog

Summary

Insufficient data verification in `UniswapNFTManager` and `BoostManager` allows attacker to break security control and use the approval the user gives to his Uniswap NFTs to `UniswapNFTManager` and `BoostManager` to steal this position funds.

In more details: `BorrowerNFT` allows any user to mint NFT's, which creates a borrower account and allows the user to control this account via the `BorrowerNFT.modify` calls. User can either call `modify` himself, or approve any address to call it on his behalf. The user can either approve address for all of his NFTs, or he can approve address for his individual NFTs. Notice: if the user approves some address to one of his NFTs, this address must not have any control over the other NFTs of the same user. The issue described in this report breaks this condition, allowing the address approved by user to one NFT to steal user funds from the other NFTs (even when not approved for them).

Suppose attacker address is approved by user to `NFT1`. And then the same user wants to use `UniswapNFTManager` and/or `BoostManager` to transfer his existing uniswap NFT position into `NFT2` (which the attacker is not approved to). Before using these managers, user must first approve his uniswap NFT position to be managed by the `UniswapNFTManager` or `BoostManager`. However, the issue is that the user can not specify which of his `NFT`s this is supposed to be used. If the user intended to deposit this position into `NFT2`, any address approved for any of his other NFTs (like attacker who is approved for `NFT1`) can use this approval to deposit this position into another NFT. So once user approves his position, attacker can immediately use this approval to deposit this into `NFT1` and immediately withdraw the funds to steal them.

Here are all the security control checks, all of them will be passed by the attacker's transaction:

1. `modify` first verifies if attacker is authorized, he is not, so `authorized` is set to `false`:

```
bool authorized = msg.sender == owner || isApprovedForAll[owner][msg.sender];
```

2. `modify` checks if attacker is approved to control the `NFT1`, he was approved, so this require passes correctly:

```
if (!authorized) require(msg.sender == getApproved[tokenId], "NOT_AUTHORIZED");
```

3. Next `modify` calls `borrower.modify` (where borrower corresponds to `NFT1`) with `UniswapNFTManager.callback` and `liquidity < 0` or `BoostManager.callback` with `action = 0`, which perform its own checks. First check in both passes (`msg.sender` is `NFT1` borrower which is registered as borrower in the `FACTORY` and the `owner` of this borrower is `BORROWER_NFT`):

```
require(FACTORY.isBorrower(msg.sender) && owner == BORROWER_NFT, "Aloe: bad
↪   caller");
```

4. Next check in both passes (uniswap NFT owner matches `NFT1` owner):

```
require(owner == UNISWAP_NFT.ownerOf(tokenId));
```

5. Finally, the liquidity is withdrawn from uniswap NFT and transferred to attacker's controlled `NFT1` (with additional `borrow` in case of `BoostManager`, which can be immediately cancelled by the attacker).

```
_withdrawFromNFT(tokenId, uint128(-liquidity), msg.sender);
borrower.uniswapDeposit(lower, upper, uint128((uint256(uint128(-liquidity)) *
↪   999) / 1000));
```

6. Since attacker has full control over `NFT1`, he can immediately withdraw the funds he just stole.

Root Cause

There is no way for `UniswapNFTManager` and `BoostManager` to distringuish which user's NFT the uniswap NFT is supposed to be for, thus any address approved for any of user's `BorrowerNFTs` can use/steal the uniswap NFT approved by the user to these managers. https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/periphery/src/managers/UniswapNFTManager.sol#L26-L54 https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/periphery/src/managers/BoostManager.sol#L37-L53

Note, there is a separate issue which might look similar (for `Permit2Manager`), however it's different as both the fix and attack path are different.

Internal pre-conditions

User has several `BorrowerNFT`s and approves attacker for only one of the NFTs, then approves his uniswap NFT to either `UniswapNFTManager` or `BoostManager`

External pre-conditions

None

Attack Path

1. Attacker listens to the user transaction, and as soon as user gives approval to any of his uniswap NFTs to either `UniswapNFTManager` or `BoostManager`, immediately call `BorrowerNFT.modify` with `NFT1`, corresponding manager and user's uniswap NFT.
2. Attacker immediately withdraws these funds from the NFT

Impact

Usage of `UniswapNFTManager` and `BoostManager` requires user to approve his uniswap NFT to these managers, but this immediately makes it possible for any address approved for any of user's `BorrowerNFT`s to steal this uniswap NFT.

PoC

N/A

Mitigation

Uniswap NFT Positions support the usage of Permit, consider using permits instead of approvals, and then also require the user to sign both permit and the `to` field so that the attacker can not use such permit with a different `BorrowerNFT`.

Discussion

nevillehuang

@panprog Can I confirm again whats the difference in root cause between this issue and in issue #7?

> In #7 the owner directly specifies to address, and the issue is that to can be modified bypassing all checks, in #8 there is not even an option to specify the to address, the approval is for any address, so it needs additional functionality. I think this makes them separate.

@WangSecurity would the above warrant is separate issues?

panprog

@nevillehuang

> Can I confirm again whats the difference in root cause between this issue and in issue #7?

In #7 the root cause is that `to` field from the permit is not signed and thus can be changed by the attacker. In this one, the user can not specify `to` address at all, so he's forced to simply approve NFT to contract, and this approval can be used by the other users. So it's somewhat similar (incorrect access control), but still different. The fix for #7 is to sign all fields, the fix for this one might be to add deposit permit with all signed fields (so make it similar to `Permit2Manager`, just for the NFT), although this will modify the current functionality. Maybe there is some other solution (like keeping manager functionality as is, but adding owner signature when spending his NFTs).

WangSecurity

I agree, even though they're similar, I agree the root causes are different.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/aloelabs/aloe-ii/pull/244

panprog

Fix review: There are still issues here.

The signature is simply a triple (borrower, tokenId, liquidity), which allows signature replay. For example:

Can be called multiple times to withdraw more liquidity than signed Can be called long time after the signature, when the owner no longer wants a particular borrower to have access to his NFT, but there is no way to retract this signature (and this can't be mitigated with removing approval, because the owner might still want to use this NFT just with a different borrower). Approval signed to BoostManager can be used in UniswapNFTManager since they use signature of the same triple of values. I suggest to add action name (keccak of contract name or address) and nonce to signed data as the bare minimum (possibly also chainId to prevent cross-chain replayability). Adding only nonce still allows to replay the signature between different managers.

Additionally, consider adding deadline to signed values, so that there is a time limit when signature can be used.

Issue M-7: `BorrowerNFT` can mint at most 65536 NFTs which is too low limit and will soon DOS for the `mint` function for all new users

Source: https://github.com/sherlock-audit/2024-06-aloe-update-judging/issues/10

Found by

panprog

Summary

`BorrowerNFT` limits NFT counter to 2 bytes or max value of 65536:

```
function _INDEX_SIZE() internal pure override returns (uint256) {
    return 2;
}
```

Once this counter value is reached (65536 NFTs minted), the `mint` call will always revert:

```
    require((totalSupply = totalSupply_ + qty) < _MAX_SUPPLY(), "MAX_SUPPLY");
...
    function _MAX_SUPPLY() internal pure returns (uint256) {
        return (1 << (_INDEX_SIZE() << 3));
    }
```

The value of 65536 is not that large, especially given that `BorrowerNFT` is specifically designed for multiple mints. For example, if each user mints 10 borrowers on average, only 6553 users can mint NFTs, and if the userbase grows larger, all new users will be denied the service due to `mint` revert. Alternatively, an attacker can simply mint all 65536 NFTs to harm the protocol and deny service to new protocol users. This will cost only gas and about 1-2 days for the attacker (10 mints per 12-seconds block).

Root Cause

`BorrowerNFT` limits NFT counter to 2 bytes (at max 65536 NFTs can be minted): https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/periphery/src/borrower-nft/BorrowerNFT.sol#L55-L57

Internal pre-conditions

65536 NFTs minted

External pre-conditions

None

Attack Path

1. 65536 NFTs minted in the protocol (either in the normal protocol operation, or by attacker)
2. Any new user tries to call `mint` which now always reverts, preventing core protocol functionality and new users joining

Impact

Core protocol functionality not working after 65536 NFTs are minted (protocol userbase exceeds about 10000 to 50000 users or attacker spending gas and 1-2 days to spam mint transactions), which sets a small hard cap limit on the number of protocol users, which can not be changed.

Note: severity is high because it's very easy and cheap for the attacker to DOS the core protocol function (`mint`) to all users.

PoC

N/A

Mitigation

Use at least 4-8 bytes for `_INDEX_SIZE`

Discussion

haydenshively

Fixed in https://github.com/aloelabs/aloe-ii/pull/241

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/aloelabs/aloe-ii/pull/241

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-8: `Lender` ERC4626 broken compliance for the `maxRedeem` and `maxWithdraw` functions, returning non-0 value which reverts in some cases

Source: https://github.com/sherlock-audit/2024-06-aloe-update-judging/issues/11

The protocol has acknowledged this issue.

Found by

panprog

Summary

README states the following about ERC4626 compliance:

> Q: Is the codebase expected to comply with any EIPs? Can there be/are there any deviations from the specification? The Lender complies with ERC4626 and EIP2612. Notes on our 4626 implementation are available here: https://coda.io/d/_dJtHvRIlBOx/Standard-Compliance_su6Wx

The link referenced provides the following compliance item in the checklist for `maxRedeem` and `maxWithdraw` functions:

> [x] MUST return the maximum amount of shares that could be transferred from owner through redeem and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).

This condition doesn't hold in the following edge case: `redeem` (and `withdraw`) reverts if the `totalSupply` after the operation is between 1 and 1e5, but this case is not handled when calculating `maxRedeem` (and `maxWithdraw`):

```
require(cache.totalSupply == 0 || cache.totalSupply > 1e5);
```

If the lender has only 1 user with deposit (who has, say, 1000000 shares) and there is a dust amount borrowed (say, 100 tokens), then `maxRedeem` will return something like 999900 shares (if `borrowBase` is 1), however if user tries to `redeem` with this amount, the operation will revert, because remaining `totalSupply` will be 100, failing the require above. The correct `maxRedeem` amount in this case is 900000 shares, as this is the max amount which can be redeemed without revert.

Root Cause

`maxRedeem` function doesn't take into account a special case when remaining `totalSupply` is between 1 and 1e5: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src/Ledger.sol#L305-L312

In such edge case the value returned by `maxRedeem` will revert the `redeem` operation: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src/Lender.sol#L200

`maxWithdraw` and `withdraw` functions simply use `maxRedeem` and `redeem`, so they have the same issue, but the root cause is the same, and once fixed, they are automatically fixed as well.

Internal pre-conditions

Only 1 user deposit into `lender`, dust amount (<1e5) borrowed from this `lender`.

External pre-conditions

None

Attack Path

1. User calls `maxRedeem` and tries to `redeem` with this amount (or alternatively, user calls `redeem` with `shares = type(uint256).max`)
2. `redeem` reverts, breaking the ERC4626 condition (redeem with amount returned by `maxRedeem` must not revert).

Impact

`Lender` is not ERC4626-compliant. Since this compliance is specifically stated in contest README, this should make this issue a valid medium.

PoC

N/A

Mitigation

Handle the special case of `totalSupply` and change `maxRedeem` amount accordingly.

Discussion

nevillehuang

Although the impact is limited, given the following judging rule, they are definitely valid.

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity. High severity will be applied only if the issue falls into the High severity category in the judging guidelines.

The READ.ME question and answer states:

> Q: Is the codebase expected to comply with any EIPs? Can there be/are there any deviations from the specification? The Lender complies with ERC4626 and EIP2612. Notes on our 4626 implementation are available here: https://coda.io/d/_dJtHvRlIBOx/Standard-Compliance_su6Wx

I believe #11, #12 and #13 can be unique issues, given they are pointing to different functionalities and involve potentially different fixes.

Issue M-9: `Lender` ERC4626 broken compliance for the `withdraw` function, in some cases withdraws more than requested amount

Source: https://github.com/sherlock-audit/2024-06-aloe-update-judging/issues/12

The protocol has acknowledged this issue.

Found by

panprog

Summary

README states the following about ERC4626 compliance:

> Q: Is the codebase expected to comply with any EIPs? Can there be/are there any deviations from the specification? The Lender complies with ERC4626 and EIP2612. Notes on our 4626 implementation are available here: https://coda.io/d/_dJtHvRlIBOx/Standard-Compliance_su6Wx

The link referenced provides the following compliance specification for `withdraw` function:

> Burns shares from owner and sends **exactly** assets of underlying tokens to receiver.

Notice that `withdraw` must send **exactly** the amount of underlying token specified, however `Lender` implementation of `withdraw` converts asset amount into shares (rounding up), then back into asset (rounding down). In some circumstances these 2 conversions might lead to a different amount from the one specified by user. For example: `inventory = 1_000_000`, `totalSupply = 990_000`, `withdraw(99 assets)` does the following:

1. Converts assets into shares:

```
    shares = previewWithdraw(amount);
...
    function previewWithdraw(uint256 assets) public view returns (uint256) {
        (, uint256 inventory, uint256 newTotalSupply) =
↪   _previewInterest(_getCache());
        return _convertToShares(assets, inventory, newTotalSupply, /* roundUp:
↪   */ true);
    }
...
    function _convertToShares(
        uint256 assets,
        uint256 inventory,
        uint256 totalSupply_,
        bool roundUp
    ) internal pure returns (uint256) {
        if (totalSupply_ == 0) return assets;
        return roundUp ? assets.mulDivUp(totalSupply_, inventory) :
 ↪  assets.mulDivDown(totalSupply_, inventory);
    }
```

```
shares = roundup(99 * 990000 / 1000000) = roundup(98.01) = 99
```

2. Convert shares back into assets:

```
    amount = _convertToAssets(shares, inventory, cache.totalSupply, /* roundUp:
 ↪  */ false);
...
    function _convertToAssets(
        uint256 shares,
        uint256 inventory,
        uint256 totalSupply_,
        bool roundUp
    ) internal pure returns (uint256) {
        if (totalSupply_ == 0) return shares;
        return roundUp ? shares.mulDivUp(inventory, totalSupply_) :
 ↪  shares.mulDivDown(inventory, totalSupply_);
    }
```

```
amount = rounddown(99 * 1000000 / 990000) = rounddown(100) = 100
```

3. This amount (100) is then sent to user (although user has requested to withdraw 99 assets).

This breaks the ERC4626 compliance for the `withdraw` function.

Root Cause

`withdraw` function simply converts asset amount to shares and uses `redeem` with this amount, which converts shares back into assets. This double-conversion can make the final asset withdrawn different from the user's requested amount due to rounding errors: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src/Lender.sol#L213-L216

Internal pre-conditions

Certain combination of total lender assets, totalSupply and user requested withdrawal amount. The higher the `assets/supply` ratio, the higher the probability of incorrect withdrawal amount. This can never happen if assets < supply, but since the lender is designed to have assets > supply, such situation can happen rather regularly, especially later in the protocol life, when assets/supply ratio is naturally higher due to growth of inventory from the borrowing rate.

External pre-conditions

None

Attack Path

User calls `withdraw` with certain amount of assets, but receives a different amount of assets (always not less than requested).

Impact

`Lender` is not ERC4626-compliant. Since this compliance is specifically stated in contest README, this should make this issue a valid medium.

PoC

Add this to `Lender.t.sol`:

```
function test_withdrawWrongAssets() public {
    // Give this test contract some shares
    deal(address(asset), address(lender), 1e6);
    lender.deposit(1e6, address(this));

    // Borrow some tokens (so that interest will actually accrue)
    hoax(address(lender.FACTORY()));
    lender.whitelist(address(this));
    lender.borrow(9e5, address(this));

    // Mock interest model
    uint256 yieldPerSecond = MAX_RATE - 1;
    vm.mockCall(
        address(lender.rateModel()),
        abi.encodeWithSelector(RateModel.getYieldPerSecond.selector, 0.5e18,
 ↪  address(lender)),
        abi.encode(yieldPerSecond)
    );

    // Now skip forward in time to make assets > total supply
    skip(1 days);

    lender.accrueInterest();
    uint256 userBalance = asset.balanceOf(address(this));

    lender.withdraw(2667, address(this), address(this));

    uint256 deltaBalance = asset.balanceOf(address(this)) - userBalance;
    console.log("withdraw(2667), actual withdrawn = %d", deltaBalance);

    vm.clearMockedCalls();
}
```

Execution console:

```
[PASS] test_withdrawWrongAssets() (gas: 493933)
Logs:
  withdraw(2667), actual withdrawn = 2668
```

Mitigation

The math for the cases in this issue makes it impossible to find an integer shares amount which will convert back to original assets amount, like:

- 98 shares -> 98 assets
- 99 shares -> 100 assets

So if 99 assets are requested, neither 98 nor 99 shares will convert to 99 assets. As such, the only fix to this problem is to handle `withdraw` function differently from `redeem`, so that the user receives his requested amount of assets exactly (burning rounded up shares amount).

Discussion

nevillehuang

Although the impact is limited, given the following judging rule, they are definitely valid.

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity. High severity will be applied only if the issue falls into the High severity category in the judging guidelines.

The READ.ME question and answer states:

> Q: Is the codebase expected to comply with any EIPs? Can there be/are there any deviations from the specification? The Lender complies with ERC4626 and EIP2612. Notes on our 4626 implementation are available here: https://coda.io/d/_dJtHvRlIBOx/Standard-Compliance_su6Wx

I believe #11, #12 and #13 can be unique issues, given they are pointing to different functionalities and involve potentially different fixes.

Issue M-10: `Lender` ERC4626 broken compliance for the `mint` function, in some cases mints more than requested amount

Source: https://github.com/sherlock-audit/2024-06-aloe-update-judging/issues/13

The protocol has acknowledged this issue.

Found by

panprog

Summary

README states the following about ERC4626 compliance:

> Q: Is the codebase expected to comply with any EIPs? Can there be/are there any deviations from the specification? The Lender complies with ERC4626 and EIP2612. Notes on our 4626 implementation are available here: https://coda.io/d/_dJtHvRlIBOx/Standard-Compliance_su6Wx

The link referenced provides the following compliance specification for `mint` function:

> Mints **exactly** shares Vault shares to receiver by depositing assets of underlying tokens.

Notice that `mint` must send **exactly** the amount of shares specified, however `Lender` implementation of `mint` converts shares amount into assets (rounding up), then back into shares (rounding down). In some circumstances these 2 conversions might lead to a different amount from the one specified by user. For example: `inventory = 999_999`, `totalSupply = 1_000_000`, `mint(999_999 shares)` does the following:

1. Converts shares into assets:

```
    amount = previewMint(shares);
...
    function previewMint(uint256 shares) public view returns (uint256) {
        (, uint256 inventory, uint256 newTotalSupply) =
↪   _previewInterest(_getCache());
        return _convertToAssets(shares, inventory, newTotalSupply, /* roundUp:
↪   */ true);
    }
...
    function _convertToAssets(
        uint256 shares,
        uint256 inventory,
        uint256 totalSupply_,
        bool roundUp
    ) internal pure returns (uint256) {
        if (totalSupply_ == 0) return shares;
        return roundUp ? shares.mulDivUp(inventory, totalSupply_) :
↪   shares.mulDivDown(inventory, totalSupply_);
    }
```

`assets = roundup(999999 * 999999 / 1000000) = roundup(999998.000001) = 999999`

2. Convert assets back into shares:

```
    shares = _convertToShares(amount, inventory, cache.totalSupply, /* roundUp:
↪   */ false);
...
    function _convertToShares(
        uint256 assets,
        uint256 inventory,
        uint256 totalSupply_,
        bool roundUp
    ) internal pure returns (uint256) {
        if (totalSupply_ == 0) return assets;
        return roundUp ? assets.mulDivUp(totalSupply_, inventory) :
↪   assets.mulDivDown(totalSupply_, inventory);
    }
```

`shares = rounddown(999999 * 1000000 / 999999) = rounddown(1000000) = 1000000`

3. This amount (1000000) is then minted to user (although user has requested to mint 999999 shares).

This breaks the ERC4626 compliance for the `mint` function.

Note: due to math, such situation happens only when `totalAssets < totalSupply`, which is a tricky situation but still possible, exact steps how it can happen are given below in Attack Path and POC.

Root Cause

`mint` function simply converts shares amount to assets amount and uses `deposit` with this amount, which converts assets back into shares. This double-conversion can make the final shares amount minted different from the user's requested amount due to rounding errors: https://github.com/sherlock-audit/2024-06-aloe-update/blob/main/aloe-ii/core/src/Lender.sol#L167-L170

The math for the cases in this issue makes it impossible to find an integer assets amount which will convert back to original shares amount, like:

- 98 assets -> 98 shares
- 99 assets -> 100 shares

So for 99 shares there is no integer amount of assets that can converts to 99 shares, which is the root cause why simple double-conversion is not enough to mint exact shares amount via deposit.

Internal pre-conditions

Certain combination of total lender assets, totalSupply and user requested mint amount. The higher the `supply/assets` ratio, the higher the probability of incorrect withdrawal amount. This can never happen if assets >= supply. Even though the protocol is designed such that assets >= supply, it's still possible to have assets < supply in some tricky edge cases.

External pre-conditions

None

Attack Path

Since protocol is designed for assets >= supply, the first part of the attack is to show scenario when assets can actually be below supply. One possible way to achieve this is to exploit borrow rounding error:

- `Lender.borrow` calculates the borrow units rounding them down:

```
units = (amount * BORROWS_SCALER) / cache.borrowIndex;
```

- However, when the borrow inventory is calculated in `_previewInterest`, the units => asset amount conversion rounds down too:

```
uint256 oldBorrows = (cache.borrowBase * cache.borrowIndex) / BORROWS_SCALER;
```

This means that if we borrow exactly 1 wei (when borrowIndex is greater than ONE, if it's exactly ONE, then division by borrowIndex is always exact due to BORROW_SCALER being ONE « 72), then the `oldBorrows` (borrow inventory) will be 0 (because our integer borrowing units amount is slightly less than exact amount of units needed for 1 wei of assets due to rounding it down).

This borrow feature allows to reduce assets while keeping totalSupply the same. So the exact steps to make assets < totalSupply:

1. Make sure that `borrowIndex` is greater than ONE (deposit, borrow, wait, repay, redeem to make it so)
2. Deposit some assets (since this is a deposit into an empty `Lender`, `totalAssets = totalSupply`)
3. borrow exactly 1 wei of assets from the `Lender` (`totalAssets` decrease by 1)
4. mint the amount of shares equal to current `totalSupply - 1` => the actual minted amount will be equal to `totalSupply` instead.

Note: while the borrow rounding error is a separate issue by itself, there might be the other ways to have assets < totalSupply.

Impact

`Lender` is not ERC4626-compliant. Since this compliance is specifically stated in contest README, this should make this issue a valid medium.

PoC

Add this to `Lender.t.sol`:

```
function test_mintWrongShares() public {
    // Give this test contract some shares
    deal(address(asset), address(lender), 1e18);
    lender.deposit(1, address(this));

    // Borrow some tokens (so that interest will actually accrue)
    hoax(address(lender.FACTORY()));
    lender.whitelist(address(this));
    lender.borrow(1, address(this));

    // Mock interest model
    uint256 yieldPerSecond = MAX_RATE - 1;
    vm.mockCall(
        address(lender.rateModel()),
        abi.encodeWithSelector(RateModel.getYieldPerSecond.selector, 0.5e18,
↪   address(lender)),
        abi.encode(yieldPerSecond)
    );

    // Now skip forward in time, but don't modify `lender` state yet
    skip(1);

    lender.repay(2, address(this)); // 2, because the debt has grown a little
    lender.redeem(1, address(this), address(this));
    // deposit again
    lender.deposit(1e6, address(this));

    console.log("Before borrow: totalAssets = %d totalSupply = %d",
↪   lender.totalAssets(), lender.totalSupply());
    lender.borrow(1, address(this)); // should reduce totalAssets so it becomes
↪   below totalSupply
    console.log("After borrow: totalAssets = %d totalSupply = %d",
↪   lender.totalAssets(), lender.totalSupply());

    // now try to mint
    uint256 userBalance = lender.balanceOf(address(this));
    lender.mint(999999, address(this));
    uint256 deltaBalance = lender.balanceOf(address(this)) - userBalance;
    console.log("mint(999999), actual minted = %d", deltaBalance);

    vm.clearMockedCalls();
}
```

Execution console:

```
[PASS] test_mintWrongShares() (gas: 572788)
Logs:
  Before borrow: totalAssets = 1000000 totalSupply = 1000000
  After borrow: totalAssets = 999999 totalSupply = 1000000
  mint(999999), actual minted = 1000000
```

Mitigation

Since it's impossible to always find assets amount which converts back exactly to a given number of shares, the only way to fix the problem is to handle `mint` function differently from `deposit`, so that the user is minted exact amount of shares, using the rounded up amount of assets.

Discussion

nevillehuang

Although the impact is limited, given the following judging rule, they are definitely valid.

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity.

> High severity will be applied only if the issue falls into the High severity category in the judging guidelines.

The READ.ME question and answer states:

> Q: Is the codebase expected to comply with any EIPs? Can there be/are there any deviations from the specification? The Lender complies with ERC4626 and EIP2612. Notes on our 4626 implementation are available here: https://coda.io/d/_dJtHvRlIBOx/Standard-Compliance_su6Wx

I believe #11, #12 and #13 can be unique issues, given they are pointing to different functionalities and involve potentially different fixes.

Disclaimers

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

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