SHERLOCK

SHERLOCK SECURITY REVIEW FOR

SHERLOCK

Prepared for:	Union Finance
Prepared by:	Sherlock
Lead Security Expert:	hyh
Dates Audited:	October 17 - October 31, 2022
Prepared on:	November 17, 2022

Introduction

Union is a member-owned credit protocol built on Ethereum where members can underwrite lines of credit to other member addresses.

Scope

```
Controller.sol
UnionLens.sol
WadRayMath.sol
AaveV3Adapter.sol
AssetManager.sol
PureTokenAdapter.sol
IAssetManager.sol
IComptroller.sol
IDai.sol
IInterestRateModel.sol
IMarketRegistry.sol
IMoneyMarketAdapter.sol
IUDai.sol
IUToken.sol
IUnionToken.sol
IUserManager.sol
AMarket3.sol
LendingPool3.sol
FixedInterestRateModel.sol
MarketRegistry.sol
UDai.sol
UToken.sol
Comptroller.sol
UserManager.sol
UserManagerDAI.sol
UserManagerERC20.sol
```

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
17	6

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

hyh
Bahurum
cccz
ctf_sec
Lambda
Jeiwan
bin2chen
obront
ak1
CodingNameKiki
seyni
Ch_301
yixxas
hansfriese
caventa
lemonmon
8olidity
dipp
GimelSec
TurnipBoy
peanuts
Picodes

Issue H-1: repayBorrow calls wrong frozen info update for overdue repayments

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/143

Found by

hyh

Summary

UToken's repayBorrow() calls frozen info update with wrong account, that end up not doing anything meaningful in the majority of cases, which leaves the stakers' account frozen info not updated, which leads to Comptroller accounting with the stale figures via debtWriteOff().

Vulnerability Detail

User facing UToken's repayBorrow() calling _repayBorrowFresh() end up calling `IUserManager(userManager).updateFrozenInfo(borrower,0)` to update frozen coin age data. However in the most cases it is pointless to call this function for the borrower as most borrowers do not have issued loans and the state of these issued loans didn't changed when this account paid own debt.

I.e. `IUserManager(userManager).updateFrozenInfo(staker,0)` is a proper call and it should be done for all borrower's stakers to reflect the changed state of their issued loans, which got healthier as borrower paid full interest or even made a prepayment with repayBorrow().

This way repayBorrow() do not update borrower's stakers frozen coin age info and this way direct UserManager's batchUpdateFrozenInfo() and Comptroller's withdrawRewards() remain the only venues for lender's account total health update.

Impact

Given that stakers are unaware of this lacking update, and assuming that frozen info update is taking place on each repayment, so frozen info is up to date, and not calling manually batchUpdateFrozenInfo() before each debtWriteOff(), they invoke Comptroller accounting update with stale `totalFrozen` in `totalStaked-totalFrozen`, diminishing `effectiveAmount` in Comptroller's inflation calculations.

As this is UNION reward miscalculation without low probability prerequisites (i.e. base functionality not working as intended), setting the severity to be high.

Code Snippet

When a borrower is overdue, `repayBorrow()->_repayBorrowFresh()` calls for

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L573-L626

```
function _repayBorrowFresh(
    address payer,
    address borrower,
    uint256 amount
) internal {
    if (!accrueInterest()) revert AccrueInterestFailed();

    uint256 interest = calculatingInterest(borrower);
    uint256 borrowedAmount = borrowBalanceStoredInternal(borrower);
    uint256 repayAmount = amount > borrowedAmount ? borrowedAmount : amount;
    if (repayAmount == 0) revert AmountZero();

    uint256 toReserveAmount;
    uint256 toRedeemableAmount;

    if (repayAmount >= interest) {
        // If the repayment amount is greater than the interest (min payment)
        bool isOverdue = checkIsOverdue(borrower);

        // Interest is split between the reserves and the uToken minters based on
        // the reserveFactorMantissa When set to WAD all the interest is paid to
↪   teh reserves.
        // any interest that isn't sent to the reserves is added to the
↪   redeemable amount
        // and can be redeemed by uToken minters.
        toReserveAmount = (interest * reserveFactorMantissa) / WAD;
        toRedeemableAmount = interest - toReserveAmount;

        // Update the total borrows to reduce by the amount of principal that has
        // been paid off
        totalBorrows -= (repayAmount - interest);

        // Update the account borrows to reflect the repayment
        accountBorrows[borrower].principal = borrowedAmount - repayAmount;
        accountBorrows[borrower].interest = 0;

        if (getBorrowed(borrower) == 0) {
            // If the principal is now 0 we can reset the last repaid block to 0.
            // which indicates that the borrower has no outstanding loans.
            accountBorrows[borrower].lastRepay = 0;
        } else {
```

```
            // Save the current block number as last repaid
            accountBorrows[borrower].lastRepay = getBlockNumber();
        }

        // Call update locked on the userManager to lock this borrowers stakers.
↪   This function
        // will revert if the account does not have enough vouchers to cover the
↪   repay amount. ie
        // the borrower is trying to repay more than is locked (owed)
        IUserManager(userManager).updateLocked(borrower, uint96(repayAmount -
↪   interest), false);

        if (isOverdue) {
            // For borrowers that are paying back overdue balances we need to
↪   update their
            // frozen balance and the global total frozen balance on the
↪   UserManager
            IUserManager(userManager).updateFrozenInfo(borrower, 0);
        }
    }
```

However, `updateFrozenInfo()->_updateFrozen()->getFrozenInfo()` aims to update overdue info from lender perspective, cycling through `vouchees` array of the issued loans:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L438-L466

```
function getFrozenInfo(address staker, uint256 pastBlocks)
    public
    view
    returns (uint256 memberTotalFrozen, uint256 memberFrozenCoinAge)
{
    uint256 overdueBlocks = uToken.overdueBlocks();
    uint256 voucheesLength = vouchees[staker].length;
    // Loop through all of the stakers vouchees sum their total
    // locked balance and sum their total memberFrozenCoinAge
    for (uint256 i = 0; i < voucheesLength; i++) {
        // Get the vouchee record and look up the borrowers voucher record
        // to get the locked amount and lastUpdate block number
        Vouchee memory vouchee = vouchees[staker][i];
        Vouch memory vouch = vouchers[vouchee.borrower][vouchee.voucherIndex];

        uint256 lastUpdated = vouch.lastUpdated;
        uint256 diff = block.number - lastUpdated;

        if (overdueBlocks < diff) {
            uint96 locked = vouch.locked;
```

```
            memberTotalFrozen += locked;
            if (pastBlocks >= diff) {
                memberFrozenCoinAge += (locked * diff);
            } else {
                memberFrozenCoinAge += (locked * pastBlocks);
            }
        }
    }
}
```

A borrower can also be a staker, but the intersection is reasonably small, i.e. calling `IUserManager(userManager).updateFrozenInfo(borrower,0)` will be a noop in the vast majority of cases.

It is also called in Comptroller via user-facing withdrawRewards() invoking _getUserInfo() that calls updateFrozenInfo():

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L231-L245

```
function withdrawRewards(address account, address token)
    external
    override
    whenNotPaused
    onlyUserManager(token)
    returns (uint256)
{
    IUserManager userManager = _getUserManager(token);

    // Lookup account state from UserManager
    (
        UserManagerAccountState memory userManagerAccountState,
        Info memory userInfo,
        uint256 pastBlocks
    ) = _getUserInfo(userManager, account, token, 0);
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L361-L384

```
function _getUserInfo(
    IUserManager userManager,
    address account,
    address token,
    uint256 futureBlocks
)
    internal
    returns (
        UserManagerAccountState memory,
```

```
        Info memory,
        uint256
    )
{
    Info memory userInfo = users[account][token];
    uint256 lastUpdatedBlock = userInfo.updatedBlock;
    if (block.number < lastUpdatedBlock) {
        lastUpdatedBlock = block.number;
    }

    uint256 pastBlocks = block.number - lastUpdatedBlock + futureBlocks;

    UserManagerAccountState memory userManagerAccountState;
    (userManagerAccountState.totalFrozen,
↪   userManagerAccountState.pastBlocksFrozenCoinAge) = userManager
        .updateFrozenInfo(account, pastBlocks);
```

withdrawRewards() is a staker reward withdrawal function.

The impact of the frozen info not being updated on borrower's repay is `comptroller.updateTotalStaked` proceeds with the stale `totalFrozen` figure:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L726-L788

```
function debtWriteOff(
    address staker,
    address borrower,
    uint96 amount
) external {
    if (amount == 0) revert AmountZero();
    uint256 overdueBlocks = uToken.overdueBlocks();
    uint256 lastRepay = uToken.getLastRepay(borrower);

    // This function is only callable by the public if the loan is overdue by
    // overdue blocks + maxOverdueBlocks. This stops the system being left with
    // debt that is overdue indefinitely and no ability to do anything about it.
    if (block.number <= lastRepay + overdueBlocks + maxOverdueBlocks) {
        if (staker != msg.sender) revert AuthFailed();
    }

    Index memory index = voucherIndexes[borrower][staker];
    if (!index.isSet) revert VoucherNotFound();
    Vouch storage vouch = vouchers[borrower][index.idx];

    if (amount > vouch.locked) revert ExceedsLocked();

    // update staker staked amount
```

```
    stakers[staker].stakedAmount -= amount;
    stakers[staker].locked -= amount;
    totalStaked -= amount;

    // update vouch trust amount
    vouch.trust -= amount;
    vouch.locked -= amount;

    // Update total frozen and member frozen. We don't want to move th
    // burden of calling updateFrozenInfo into this function as it is quite
    // gas intensive. Instead we just want to remove the amount that was
    // frozen which is now being written off. However, it is possible that
    // member frozen has not been updated prior to calling debtWriteOff and
    // the amount being written off could be greater than the amount frozen.
    // To avoid an underflow here we need to check this condition
    uint256 stakerFrozen = memberFrozen[staker];
    if (amount > stakerFrozen) {
        // The amount being written off is more than the amount that has
        // been previously frozen for this staker. Reset their frozen stake
        // to zero and adjust totalFrozen
        memberFrozen[staker] = 0;
        totalFrozen -= stakerFrozen;
    } else {
        totalFrozen -= amount;
        memberFrozen[staker] -= amount;
    }

    if (vouch.trust == 0) {
        cancelVouch(staker, borrower);
    }

    // Notify the AssetManager and the UToken market of the debt write off
    // so they can adjust their balances accordingly
    IAssetManager(assetManager).debtWriteOff(stakingToken, uint256(amount));
    uToken.debtWriteOff(borrower, uint256(amount));

    comptroller.updateTotalStaked(stakingToken, totalStaked - totalFrozen);

    emit LogDebtWriteOff(msg.sender, borrower, uint256(amount));
}
```

Tool used

Manual Review

Recommendation

Consider calling it for borrower's staker instead:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L621-L625

```
            if (isOverdue) {
                // For borrowers that are paying back overdue balances we need to
↪   update their
                // frozen balance and the global total frozen balance on the
↪   UserManager
-               IUserManager(userManager).updateFrozenInfo(borrower, 0);
+       ... // cycle all stakers of the borrower
+                   IUserManager(userManager).updateFrozenInfo(staker, 0);
            }
```

An alternative to this is to call updateFrozenInfo() in updateLocked() as it implements closely related logic.

Discussion

kingjacob

This is as designed, The contract assumes a minimal level of activity or a keeper to keep reward calcs up to date.

dmitriia

It's not about updating one staker or all of them, it's updating **staker**, not the borrower: calling `IUserManager(userManager).updateFrozenInfo(borrower,0)` is just pointless, most borrowers will not have stakes of their own, and those stakes, if exist, aren't updated on repayBorrow.

In other words:

- Bob opens trust for Alice
- Alice borrows, then repays
- `updateFrozenInfo` is called for Alice, while she doesn't have any stakes and frozen info is irrelevant for her
- `updateFrozenInfo` should be called for Bob
- ideally for all Alice stakers, but it's design question indeed, the issue is about calling for Bob instead of Alice

kingjacob

@dmitriia gotcha, youre probably right on this.

Issue H-2: Stakers will lose their rewards as updateLocked() updates only the first active vouches until there is a pre-payment

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/135

Found by

hyh

Summary

Whenever Bob the borrower has several lenders and pays only interest without pre-payments, only the first of his stakers will have good health of his loan accounted for by keeping `vouch.lastUpdated` current, all the rest will lose the portion of rewards associated with Bob's loan as it will be counted as overdue, despite the fact that Bob has paid the **full** total interest without a single delay.

Vulnerability Detail

When a borrower pays interest in full, UserManager's updateLocked() is called by UToken's _repayBorrowFresh() to mark the active vouches in the Bob's `vouchers` array with `vouch.lastUpdated=uint64(block.number)`. However, updateLocked() logic performs iterations over the array only up to the notional prepayment accounting point.

I.e. if Bob paid only full interest, not performing any notional prepayments, updateLocked() updates only the very first active vouch, as updateLocked() was called with `amount=repayAmount-interest=0`.

Moreover, if Bob does some prepayment, the cycle continues only to mark enough vouches to account for that prepayment, i.e. it stops once the prepayment is done.

The core issue is that updateLocked() logic is made for this prepayments accounting, while it is also used for marking non-overdue vouches.

For that matter the `if(remaining<=0)break` is in fact incorrect as the condition for cycle ending as for marking of the timely payment it doesn't matter how much Bob has prepaid, and all his active vouches need to be updated as Bob had payed interest for all of them, which was tracked in _repayBorrowFresh() by `calculatingInterest(Bob)`, which is total current interest for all Bob's loans.

Impact

All Bob's lenders except the first one (or maybe first ones if there was some prepayment) will lose the corresponding part of their rewards despite the fact that Bob keeps his loan in the perfect health, not missing a single payment.

As UNION rewards is the main venue of staker profit from the system, this is a material loss for all of them. There are no any additional conditions, i.e. the loss of rewards happens in a going concern state, while even full interest payments without any prepayments is one of the base use cases for any borrower.

The total impact is massive reward loss without low probability preconditions, so setting the severity to be **high**.

Code Snippet

Bob pays the interest by calling repayBorrow() -> _repayBorrowFresh():

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L561-L563

```
function repayBorrow(address borrower, uint256 repayAmount) external override
↪   whenNotPaused nonReentrant {
    _repayBorrowFresh(msg.sender, borrower, repayAmount);
}
```

_repayBorrowFresh() calculates the total `interest=calculatingInterest(borrower)` due to Bob and calls `IUserManager(userManager).updateLocked(borrower,uint96(repayAmount-interest),false)` if he has paid this interest in full:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L573-L626

```
function _repayBorrowFresh(
    address payer,
    address borrower,
    uint256 amount
) internal {
    if (!accrueInterest()) revert AccrueInterestFailed();

    uint256 interest = calculatingInterest(borrower);
    uint256 borrowedAmount = borrowBalanceStoredInternal(borrower);
    uint256 repayAmount = amount > borrowedAmount ? borrowedAmount : amount;
    if (repayAmount == 0) revert AmountZero();

    uint256 toReserveAmount;
    uint256 toRedeemableAmount;
```

```
    if (repayAmount >= interest) {
        // If the repayment amount is greater than the interest (min payment)
        bool isOverdue = checkIsOverdue(borrower);

        // Interest is split between the reserves and the uToken minters based on
        // the reserveFactorMantissa When set to WAD all the interest is paid to
↪   teh reserves.
        // any interest that isn't sent to the reserves is added to the
↪   redeemable amount
        // and can be redeemed by uToken minters.
        toReserveAmount = (interest * reserveFactorMantissa) / WAD;
        toRedeemableAmount = interest - toReserveAmount;

        // Update the total borrows to reduce by the amount of principal that has
        // been paid off
        totalBorrows -= (repayAmount - interest);

        // Update the account borrows to reflect the repayment
        accountBorrows[borrower].principal = borrowedAmount - repayAmount;
        accountBorrows[borrower].interest = 0;

        if (getBorrowed(borrower) == 0) {
            // If the principal is now 0 we can reset the last repaid block to 0.
            // which indicates that the borrower has no outstanding loans.
            accountBorrows[borrower].lastRepay = 0;
        } else {
            // Save the current block number as last repaid
            accountBorrows[borrower].lastRepay = getBlockNumber();
        }

        // Call update locked on the userManager to lock this borrowers stakers.
↪   This function
        // will revert if the account does not have enough vouchers to cover the
↪   repay amount. ie
        // the borrower is trying to repay more than is locked (owed)
        IUserManager(userManager).updateLocked(borrower, uint96(repayAmount -
↪   interest), false);

        if (isOverdue) {
            // For borrowers that are paying back overdue balances we need to
↪   update their
            // frozen balance and the global total frozen balance on the
↪   UserManager
            IUserManager(userManager).updateFrozenInfo(borrower, 0);
        }
    }
```


If Bob has many vouches and pays full interest, but nothing else, before minimum payment period runs out every time, **only** his first vouch will be marked as payed by updating `vouch.lastUpdated=uint64(block.number)`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L828-L846

```
} else {
    // Look up how much this vouch has locked. If it is 0 then
    // continue to the next voucher. Then calculate the amount to
    // unlock which is the min of the vouches lock and what is
    // remaining to unlock
    uint96 locked = vouch.locked;
    if (locked == 0) continue;
    innerAmount = _min(locked, remaining);

    // Update the stored locked values and last updated block
    stakers[vouch.staker].locked -= innerAmount;
    vouch.locked -= innerAmount;
    vouch.lastUpdated = uint64(block.number);
}

remaining -= innerAmount;
// If there is no remaining amount to lock/unlock
// we can stop looping through vouchers
if (remaining <= 0) break;
```

I.e. the `if(remaining<=0)break` condition is preventing all other Bob's vouches to be updated as healthy, damaging the corresponding lenders.

Reward multiplier is based on the frozen coin age, which is calculated with the help of `vouch.lastUpdated` of all staker's vouches:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L432-L466

```
/**
 *  @dev Get frozen coin age
 *  @param  staker Address of staker
 *  @param  pastBlocks Number of blocks past to calculate coin age from
 *          coin age = min(block.number - lastUpdated, pastBlocks) * amount
 */
function getFrozenInfo(address staker, uint256 pastBlocks)
    public
    view
    returns (uint256 memberTotalFrozen, uint256 memberFrozenCoinAge)
{
    uint256 overdueBlocks = uToken.overdueBlocks();
```

```
    uint256 voucheesLength = vouchees[staker].length;
    // Loop through all of the stakers vouchees sum their total
    // locked balance and sum their total memberFrozenCoinAge
    for (uint256 i = 0; i < voucheesLength; i++) {
        // Get the vouchee record and look up the borrowers voucher record
        // to get the locked amount and lastUpdate block number
        Vouchee memory vouchee = vouchees[staker][i];
        Vouch memory vouch = vouchers[vouchee.borrower][vouchee.voucherIndex];

        uint256 lastUpdated = vouch.lastUpdated;
        uint256 diff = block.number - lastUpdated;

        if (overdueBlocks < diff) {
            uint96 locked = vouch.locked;
            memberTotalFrozen += locked;
            if (pastBlocks >= diff) {
                memberFrozenCoinAge += (locked * diff);
            } else {
                memberFrozenCoinAge += (locked * pastBlocks);
            }
        }
    }
}
```

Tool used

Manual Review

Recommendation

One straightforward way to circumvent this is to run the full cycle across all Bob's vouches, setting `vouch.lastUpdated=uint64(block.number)` for all of them and not exiting when `if(remaining<=0)break`. Current partial redemption logic remains as it is, only doing nothing when `remaining` is exhausted, i.e. to be placed into a `if(remaining>0)` block.

Another option is to base the reward multiplier calculation not on the `vouch.lastUpdated`, but on the health of each borrower of the staker, i.e. to base total frozen on borrower's `lastRepay` instead of voucher's `lastUpdate`.

Borrower's `lastRepay` is the only indicator for the checkIsOverdue() and debtWriteOff():

checkIsOverdue() and debtWriteOff() use getLastRepay():

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L371-L382

```
/**
 *  @dev Check if the member's loan is overdue
 *  @param account Member address
 *  @return isOverdue
 */
function checkIsOverdue(address account) public view override returns (bool
↪  isOverdue) {
    if (getBorrowed(account) != 0) {
        uint256 lastRepay = getLastRepay(account);
        uint256 diff = getBlockNumber() - lastRepay;
        isOverdue = overdueBlocks < diff;
    }
}
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L726-L740

```
function debtWriteOff(
    address staker,
    address borrower,
    uint96 amount
) external {
    if (amount == 0) revert AmountZero();
    uint256 overdueBlocks = uToken.overdueBlocks();
    uint256 lastRepay = uToken.getLastRepay(borrower);

    // This function is only callable by the public if the loan is overdue by
    // overdue blocks + maxOverdueBlocks. This stops the system being left with
    // debt that is overdue indefinitely and no ability to do anything about it.
    if (block.number <= lastRepay + overdueBlocks + maxOverdueBlocks) {
        if (staker != msg.sender) revert AuthFailed();
    }
```

getLastRepay() gets `accountBorrows[].lastRepay`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L367-L369

```
function getLastRepay(address account) public view override returns (uint256) {
    return accountBorrows[account].lastRepay;
}
```

To unify the logic and simplify vouch workflow, `lastRepay` of staker's borrowers can be also used for the frozen age calculations, for example:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-cont

racts/contracts/user/UserManager.sol#L432-L466

```
    /**
     *  @dev Get frozen coin age
     *  @param  staker Address of staker
     *  @param  pastBlocks Number of blocks past to calculate coin age from
     *          coin age = min(block.number - lastUpdated, pastBlocks) * amount
     */
    function getFrozenInfo(address staker, uint256 pastBlocks)
        public
        view
        returns (uint256 memberTotalFrozen, uint256 memberFrozenCoinAge)
    {
        uint256 overdueBlocks = uToken.overdueBlocks();
        uint256 voucheesLength = vouchees[staker].length;
        // Loop through all of the stakers vouchees sum their total
        // locked balance and sum their total memberFrozenCoinAge
        for (uint256 i = 0; i < voucheesLength; i++) {
            // Get the vouchee record and look up the borrowers voucher record
            // to get the locked amount and lastUpdate block number
            Vouchee memory vouchee = vouchees[staker][i];
            Vouch memory vouch =
 ↪  vouchers[vouchee.borrower][vouchee.voucherIndex];


-           uint256 lastUpdated = vouch.lastUpdated;
+           uint256 lastUpdated = uToken.getLastRepay(vouchee);
+       if (vouch.locked == 0 || lastUpdated == 0) continue;  // should be zero
 ↪  simultaneously, but we need to skip either way
            uint256 diff = block.number - lastUpdated;

            if (overdueBlocks < diff) {
                uint96 locked = vouch.locked;
                memberTotalFrozen += locked;
                if (pastBlocks >= diff) {
                    memberFrozenCoinAge += (locked * diff);
                } else {
                    memberFrozenCoinAge += (locked * pastBlocks);
                }
            }
        }
    }
```

Discussion

kingjacob

Dupe of #99 ?

dmitriia

> Dupe of #99 ?

Looks like #99 and #74 are dupes, this one is actually a bit more systematic, with similar kind of outcome without the need of any direct attack

Issue H-3: repayBorrow is inaccessible by overdue borrowers

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/133

Found by

hyh, CodingNameKiki, Lambda, seyni

Summary

User facing UToken's `repayBorrow()->_repayBorrowFresh()` will be reverted when borrower is overdue, making it permanently impossible to pay the debt within the system.

Vulnerability Detail

UToken's _repayBorrowFresh() will attempt to update frozen info when borrower is overdue, but the corresponding function of UserManager, updateFrozenInfo(), is `onlyComptroller`, so the call will be reverted.

As repayBorrow() with payment exceeding the total interest is the only way for a borrower to repay their loan either partially or fully, for the lenders of this borrower this means that the debt will never be repaid as there is no technical possibility to do it. I.e. all notional of all debts whose borrowers are overdue are permanently frozen.

Such borrowers are stuck with the overdue status and can only pay partial interest payments.

Impact

The total impact is permanent fund freeze for the lenders of any overdue borrowers.

The only prerequisite is borrower turning overdue, which is a pretty common business case, so setting the severity to be **high**.

Code Snippet

_repayBorrowFresh() calls `IUserManager(userManager).updateFrozenInfo(borrower, 0)` when `checkIsOverdue(borrower)` is true:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L621-L625

```
if (isOverdue) {
    // For borrowers that are paying back overdue balances we need to update
    ↪ their
    // frozen balance and the global total frozen balance on the UserManager
    IUserManager(userManager).updateFrozenInfo(borrower, 0);
}
```

updateFrozenInfo() is `onlyComptroller`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L881-L883

```
function updateFrozenInfo(address staker, uint256 pastBlocks) external
    ↪ onlyComptroller returns (uint256, uint256) {
    return _updateFrozen(staker, pastBlocks);
}
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L305-L308

```
modifier onlyComptroller() {
    if (address(comptroller) != msg.sender) revert AuthFailed();
    _;
}
```

This means that any overdue borrower is stuck, being unable neither to remove overdue status, nor to repay the debt as it is the very same piece of logic, that reverts whenever called given that borrower is overdue:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L573-L626

```
function _repayBorrowFresh(
    address payer,
    address borrower,
    uint256 amount
) internal {
    if (!accrueInterest()) revert AccrueInterestFailed();

    uint256 interest = calculatingInterest(borrower);
    uint256 borrowedAmount = borrowBalanceStoredInternal(borrower);
    uint256 repayAmount = amount > borrowedAmount ? borrowedAmount : amount;
    if (repayAmount == 0) revert AmountZero();

    uint256 toReserveAmount;
    uint256 toRedeemableAmount;
```

```
    if (repayAmount >= interest) {
        // If the repayment amount is greater than the interest (min payment)
        bool isOverdue = checkIsOverdue(borrower);

        // Interest is split between the reserves and the uToken minters based on
        // the reserveFactorMantissa When set to WAD all the interest is paid to
↪   teh reserves.
        // any interest that isn't sent to the reserves is added to the
↪   redeemable amount
        // and can be redeemed by uToken minters.
        toReserveAmount = (interest * reserveFactorMantissa) / WAD;
        toRedeemableAmount = interest - toReserveAmount;

        // Update the total borrows to reduce by the amount of principal that has
        // been paid off
        totalBorrows -= (repayAmount - interest);

        // Update the account borrows to reflect the repayment
        accountBorrows[borrower].principal = borrowedAmount - repayAmount;
        accountBorrows[borrower].interest = 0;

        if (getBorrowed(borrower) == 0) {
            // If the principal is now 0 we can reset the last repaid block to 0.
            // which indicates that the borrower has no outstanding loans.
            accountBorrows[borrower].lastRepay = 0;
        } else {
            // Save the current block number as last repaid
            accountBorrows[borrower].lastRepay = getBlockNumber();
        }

        // Call update locked on the userManager to lock this borrowers stakers.
↪   This function
        // will revert if the account does not have enough vouchers to cover the
↪   repay amount. ie
        // the borrower is trying to repay more than is locked (owed)
        IUserManager(userManager).updateLocked(borrower, uint96(repayAmount -
↪   interest), false);

        if (isOverdue) {
            // For borrowers that are paying back overdue balances we need to
↪   update their
            // frozen balance and the global total frozen balance on the
↪   UserManager
            IUserManager(userManager).updateFrozenInfo(borrower, 0);
        }
    }
```


I.e. once borrower is overdue the only way to remove that status is to run `accountBorrows[borrower].lastRepay=getBlockNumber()` line of _repayBorrowFresh(), as there are no other ways to do that in the system.

In the same time when the borrower is overdue the code will always revert on `IUserManager(userManager).updateFrozenInfo(borrower,0)` call as it's `onlyComptroller`.

Tool used

Manual Review

Recommendation

Consider updating it to be either UToken or Comptroller as both contracts use it:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L881-L883

```
-   function updateFrozenInfo(address staker, uint256 pastBlocks) external
 ↪  onlyComptroller returns (uint256, uint256) {
+   function updateFrozenInfo(address staker, uint256 pastBlocks) external
 ↪  returns (uint256, uint256) {
+       if (address(uToken) != msg.sender || address(comptroller) != msg.sender)
 ↪  revert AuthFailed();
        return _updateFrozen(staker, pastBlocks);
    }
```

Issue H-4: Loan can be written off by anybody before overdue delay expires

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/115

Found by

Bahurum

Summary

When a borrower takes a second loan after a loan that has been written off, this second loan can be written off instantly by any other member due to missing update of last repay block, leaving the staker at a loss.

Vulnerability Detail

1. A staker stakes and vouches a borrower
2. the borrower borrows calling `UToken:borrow`: `accountBorrows[borrower].lastRepay` is updated with the current block number
3. the staker writes off the entire debt of the borrower calling `UserManager:debtWriteOff`. In the internal call to `UToken:debtWriteOff` the principal is set to zero but `accountBorrows[borrower].lastRepay` is not updated
4. 90 days pass and a staker vouches for the same borrower
5. the borrower borrows calling `UToken:borrow`: `accountBorrows[borrower].lastRepay` is not set to the current block since non zero and stays to the previous value.
6. `accountBorrows[borrower].lastRepay` is now old enough to allow the check in `UserManager:debtWriteOff` at line 738 to pass. The debt is written off by any other member immediatly after the loan is given. The staker looses the staked amount immediatly.

```
if (block.number <= lastRepay + overdueBlocks + maxOverdueBlocks) {
    if (staker != msg.sender) revert AuthFailed();
}
```

7. The last repay block is still stale and a new loan can be taken and written off immediatly many times as long as stakers are trusting the borrower

Note that this can be exploited maliciously by the borrower, who can continously ask for loans and then write them off immediatly.

Impact

The staker of the loan looses the staked amount well before the overdue delay is expired

Code Snippet

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L666-L672

```
function debtWriteOff(address borrower, uint256 amount) external override
↪   whenNotPaused onlyUserManager {
    uint256 oldPrincipal = getBorrowed(borrower);
    uint256 repayAmount = amount > oldPrincipal ? oldPrincipal : amount;

    accountBorrows[borrower].principal = oldPrincipal - repayAmount;
    totalBorrows -= repayAmount;
}
```

Tool used

Manual Review

Recommendation

Reset `lastRepay` for the borrower to 0 when the debt is written off completely

```
    function debtWriteOff(address borrower, uint256 amount) external override
↪   whenNotPaused onlyUserManager {
        uint256 oldPrincipal = getBorrowed(borrower);
        uint256 repayAmount = amount > oldPrincipal ? oldPrincipal : amount;

+       if (oldPrincipal == repayAmount) accountBorrows[borrower].lastRepay = 0;
        accountBorrows[borrower].principal = oldPrincipal - repayAmount;
        totalBorrows -= repayAmount;
    }
```

Issue H-5: Staker rewards can be gathered with maximal multiplier no matter how borrowers are overdue

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/74

Found by

hyh, cccz

Summary

Frozen coin age of an account is based on the `vouch.lastUpdated`, which is set to the current block not only on initial borrowing and debt full interest payment or notional repayment, but also on any repetitive borrowing from a vouch.

Using this a staker can collude with their borrowers, asking them to borrow a minimal amount, so staker's `memberTotalFrozen` end up being zero and maximal rewards can be obtained immediately with UserManager's withdrawRewards().

Vulnerability Detail

Bob the staker whose borrowers are overdue can ask them to borrow min amount as soon as they can (1 DAI), then claim the rewards, which will be with maximal `2x` multiplier.

Bob can do it each time he wants to gather the rewards (say before he unstakes), effectively surpassing the reward multiplier logic altogether.

Another theoretical way to game the rewards logic is to lend to self, but this require paying interest, that isn't low. On the other side minimal borrow amount is kept low, now staying at `1.0DAI`, to promote protocol adoption.

This way, Bob and his borrowers excess costs are negligible (gas and min borrowings), while benefit of obtaining max rewards are material.

I.e. Bob, being a rational user, on each reward withdrawal can ask the vouchees to borrow minimal amount (Bob will update trust for that end, if needed) instead of repaying anything, this way removing the penalty. Bob will obtain the rewards with the maximal multiplier.

Impact

Bob gains bloated reward issuance and steals from all UNION token holders by having its supply diluted.

As this can be done without any major prerequisites, setting the severity to be **high**.

Code Snippet

UNION tokens issuance is the main staker profit mechanics, while stake utilization is the core metric for reward issuance rate calculations:

https://docs.union.finance/protocol-overview/plain-english-overview#stakers-earning-union-from-comptroller

```
Stake utilization is meant to give control of the protocol to those who vouch for
↪  people who borrow and dont default. And the 0.75 is to allow non-members a
↪  non financial means but still sybil resistant way to acquire the UNION. The
↪  curve is upgradeable without requiring a fork or a withdrawal of the dai
↪  staked. All of this is handled by the {Comptroller.sol Contract}
```

UserManager's updateLocked() sets `vouchers[borrower][i].lastUpdated` to the current block both on borrowing and repaying:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L800-L841

```
function updateLocked(
    address borrower,
    uint96 amount,
    bool lock
) external onlyMarket {
    uint96 remaining = amount;

    for (uint256 i = 0; i < vouchers[borrower].length; i++) {
        Vouch storage vouch = vouchers[borrower][i];
        uint96 innerAmount;

        if (lock) {
            // Look up the staker and determine how much unlock stake they
            // have available for the borrower to borrow. If there is 0
            // then continue to the next voucher in the array
            uint96 stakerLocked = stakers[vouch.staker].locked;
            uint96 stakerStakedAmount = stakers[vouch.staker].stakedAmount;
            uint96 availableStake = stakerStakedAmount - stakerLocked;
            uint96 lockAmount = _min(availableStake, vouch.trust - vouch.locked);
            if (lockAmount == 0) continue;

            // Calculate the amount to add to the lock then
            // add the extra amount to lock to the stakers locked amount
            // and also update the vouches locked amount and lastUpdated block
            innerAmount = _min(remaining, lockAmount);
            stakers[vouch.staker].locked = stakerLocked + innerAmount;
            vouch.locked += innerAmount;
            vouch.lastUpdated = uint64(block.number);
```

```
        } else {
            // Look up how much this vouch has locked. If it is 0 then
            // continue to the next voucher. Then calculate the amount to
            // unlock which is the min of the vouches lock and what is
            // remaining to unlock
            uint96 locked = vouch.locked;
            if (locked == 0) continue;
            innerAmount = _min(locked, remaining);

            // Update the stored locked values and last updated block
            stakers[vouch.staker].locked -= innerAmount;
            vouch.locked -= innerAmount;
            vouch.lastUpdated = uint64(block.number);
        }
```

getFrozenInfo() uses `vouchers[vouchee.borrower][vouchee.voucherIndex].lastUpdated` for calculating what share of the total staker's lending business with the system is overdue:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L438-L454

```
function getFrozenInfo(address staker, uint256 pastBlocks)
    public
    view
    returns (uint256 memberTotalFrozen, uint256 memberFrozenCoinAge)
{
    uint256 overdueBlocks = uToken.overdueBlocks();
    uint256 voucheesLength = vouchees[staker].length;
    // Loop through all of the stakers vouchees sum their total
    // locked balance and sum their total memberFrozenCoinAge
    for (uint256 i = 0; i < voucheesLength; i++) {
        // Get the vouchee record and look up the borrowers voucher record
        // to get the locked amount and lastUpdate block number
        Vouchee memory vouchee = vouchees[staker][i];
        Vouch memory vouch = vouchers[vouchee.borrower][vouchee.voucherIndex];

        uint256 lastUpdated = vouch.lastUpdated;
        uint256 diff = block.number - lastUpdated;
```

I.e. as a result getFrozenInfo() will treat constantly borrowing account as not frozen.

Tool used

Manual Review

Recommendation

Consider either basing total frozen on borrower's `lastRepay` instead of voucher's `lastUpdate` (as it described in another issue) or removing the `lastUpdate` renewal on a repetitive borrow.

I.e. update of the indicator that is used in the total frozen calculation on a repetitive borrow is the issue here as it neither loan start nor full interest / partial notional prepayment and shouldn't reset the frozen counter.

In other words frozen age is overdue analogue with regard to lender's account health. Repetitive borrow do not reduce this overdue, but is counted towards it.

As `lastUpdate` is used in account frozen statistics calculation only, the simplest solution is to change `lastUpdate` on the notional repay and initial borrowing from the vouch only:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L811-L828

```
            if (lock) {
                // Look up the staker and determine how much unlock stake they
                // have available for the borrower to borrow. If there is 0
                // then continue to the next voucher in the array
                uint96 stakerLocked = stakers[vouch.staker].locked;
                uint96 stakerStakedAmount = stakers[vouch.staker].stakedAmount;
                uint96 availableStake = stakerStakedAmount - stakerLocked;
+               uint96 vouchLocked = vouch.locked;
-               uint96 lockAmount = _min(availableStake, vouch.trust -
 ↪  vouch.locked);
+               uint96 lockAmount = _min(availableStake, vouch.trust -
 ↪  vouchLocked);
                if (lockAmount == 0) continue;

                // Calculate the amount to add to the lock then
                // add the extra amount to lock to the stakers locked amount
                // and also update the vouches locked amount and lastUpdated
 ↪  block
                innerAmount = _min(remaining, lockAmount);
                stakers[vouch.staker].locked = stakerLocked + innerAmount;
+               vouch.locked = vouchLocked + innerAmount;
+               if (vouchLocked == 0) vouch.lastUpdated = uint64(block.number);
-               vouch.locked += innerAmount;
-               vouch.lastUpdated = uint64(block.number);
            }
```

Notice, that this example solution will suffice if the issue is treated in isolation, but given other issues using `lastRepay` is preferred.

Discussion

kingjacob

Dupe of #99

Issue H-6: UNION rewards issuance can be maximized without providing credit

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/49

Found by

hyh

Summary

UNION rewards accounting is run on user direct request and stake/unstake only, and due to this can be gamed to obtain maximal rewards without any involvement in UNION, and in a risk-free way.

Vulnerability Detail

Reward calculation currently is being invoked only as a part of user-facing stake(), unstake() and withdrawRewards(). Notice that withdrawRewards() calculates the rewards for `msg.sender`, so it cannot be triggered automatically by a script for all active users to refresh their accounting. This way UNION rewards are calculated only per user request and there are no updates on key account state changing operations.

This combination allows a malicious user to receive rewards with maximal `2x` multiplier without participating in UNION, particularly without bearing any credit risk or interest expenses.

Say Bob the staker wants to maximize UNION rewards and minimize the corresponding risk. Bob stakes and do nothing else for a prolonged period, say 2 years. Then he becomes a member, loans to himself the full stake via another account and immediately calls withdrawRewards(), obtaining rewards for the whole 2 year period, his full stake, with the maximal multiplier, i.e. with `_getRewardsMultiplier` of `2`.

Impact

Bloated UNION rewards emissions attacker initiates dilute holdings of the honest members who do not time their withdrawRewards() calls to the points of reward multiplier maximization.

This is monetary loss for all UNION holders (as total emission is increased), and particularly for the stakers who actually use the system (as specifically new emission is increased), i.e. in result the attacker steals from all holders/stakers by inflating UNION emission.

Setting severity to be high as there are no specific preconditions for the attack, it can be carried out by any UNION staker.

Code Snippet

The only function that updates the rewards due to a particular user is Comptroller's _calculateRewardsByBlocks().

_calculateRewardsByBlocks() calculates the rewards based on the current state of a user:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L396-L429

```
function _calculateRewardsByBlocks(
    address account,
    address token,
    uint256 pastBlocks,
    Info memory userInfo,
    UserManagerState memory userManagerState,
    UserManagerAccountState memory userManagerAccountState
) internal view returns (uint256) {
    IUserManager userManagerContract = _getUserManager(token);

    // Lookup account state from UserManager
    userManagerAccountState.totalStaked =
↪   userManagerContract.getStakerBalance(account);
    userManagerAccountState.totalLocked =
↪   userManagerContract.getTotalLockedStake(account);
    userManagerAccountState.isMember =
↪   userManagerContract.checkIsMember(account);

    uint256 inflationIndex = _getRewardsMultiplier(
        userManagerAccountState.totalStaked,
        userManagerAccountState.totalLocked,
        userManagerAccountState.totalFrozen,
        userManagerAccountState.isMember
    );

    return
        userInfo.accrued +
        _calculateRewards(
            account,
            token,
            userManagerState.totalStaked,
            userManagerAccountState.totalStaked,
            userManagerAccountState.pastBlocksFrozenCoinAge,
            pastBlocks,
```

```
            inflationIndex
        );
}
```

Specifically, `inflationIndex` is based solely on the current state of user's account:

```
inflationIndex = _getRewardsMultiplier(
            userManagerAccountState.totalStaked,
            userManagerAccountState.totalLocked,
            userManagerAccountState.totalFrozen,
            userManagerAccountState.isMember
        )
```

_calculateRewards() multiplies the whole `(curInflationIndex-startInflationIndex).wadMul(effectiveStakeAmount)` by the `inflationIndex`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L443-L466

```
function _calculateRewards(
    address account,
    address token,
    uint256 totalStaked,
    uint256 userStaked,
    uint256 frozenCoinAge,
    uint256 pastBlocks,
    uint256 inflationIndex
) internal view returns (uint256) {
    uint256 startInflationIndex = users[account][token].inflationIndex;
    if (userStaked * pastBlocks < frozenCoinAge) revert FrozenCoinAge();

    if (userStaked == 0 || totalStaked == 0 || startInflationIndex == 0 ||
↪   pastBlocks == 0) {
        return 0;
    }

    uint256 effectiveStakeAmount = (userStaked * pastBlocks - frozenCoinAge) /
↪   pastBlocks;

    uint256 curInflationIndex = _getInflationIndexNew(totalStaked, pastBlocks);

    if (curInflationIndex < startInflationIndex) revert InflationIndexTooSmall();

    return (curInflationIndex -
↪   startInflationIndex).wadMul(effectiveStakeAmount).wadMul(inflationIndex);
}
```

`inflationIndex=_getRewardsMultiplier(...)` is based on the current state of a user:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L523-L543

```
function _getRewardsMultiplier(
    uint256 userStaked,
    uint256 lockedStake,
    uint256 totalFrozen_,
    bool isMember_
) internal pure returns (uint256) {
    if (isMember_) {
        if (userStaked == 0 || totalFrozen_ >= lockedStake) {
            return memberRatio;
        }

        uint256 effectiveLockedAmount = lockedStake - totalFrozen_;
        uint256 effectiveStakeAmount = userStaked - totalFrozen_;

        uint256 lendingRatio =
↪   effectiveLockedAmount.wadDiv(effectiveStakeAmount);

        return lendingRatio + memberRatio;
    } else {
        return nonMemberRatio;
    }
}
```

_calculateRewardsByBlocks() is invoked in Comptroller only as a part of two call sequences:

calculateRewards() -> calculateRewardsByBlocks() -> _calculateRewardsByBlocks(),

withdrawRewards() -> _calculateRewardsByBlocks().

Comptroller's calculateRewards() and calculateRewardsByBlocks() are public views, not used elsewhere:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L175-L211

```
/**
 *  @dev Calculate unclaimed rewards based on blocks
 *  @param account User address
 *  @param token Staking token address
 *  @param futureBlocks Number of blocks in the future
 *  @return Unclaimed rewards
 */
function calculateRewardsByBlocks(
```

```
    address account,
    address token,
    uint256 futureBlocks
) public view override returns (uint256) {
    IUserManager userManager = _getUserManager(token);

    // Lookup account stataddress accounte from UserManager
    (
        UserManagerAccountState memory userManagerAccountState,
        Info memory userInfo,
        uint256 pastBlocks
    ) = _getUserInfoView(userManager, account, token, futureBlocks);

    // Lookup global state from UserManager
    UserManagerState memory userManagerState = _getUserManagerState(userManager);

    return
        _calculateRewardsByBlocks(account, token, pastBlocks, userInfo,
↪   userManagerState, userManagerAccountState);
}

/**
 *  @dev Calculate currently unclaimed rewards
 *  @param account Account address
 *  @param token Staking token address
 *  @return Unclaimed rewards
 */
function calculateRewards(address account, address token) external view override
↪   returns (uint256) {
    return calculateRewardsByBlocks(account, token, 0);
}
```

This way the only rewards accounting update is Comptroller's withdrawRewards(), which is `onlyUserManager`, and is called via user facing UserManager's stake(), unstake() and withdrawRewards():

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L231-L257

```
function withdrawRewards(address account, address token)
    external
    override
    whenNotPaused
    onlyUserManager(token)
    returns (uint256)
{
    IUserManager userManager = _getUserManager(token);
```

```
    // Lookup account state from UserManager
    (
        UserManagerAccountState memory userManagerAccountState,
        Info memory userInfo,
        uint256 pastBlocks
    ) = _getUserInfo(userManager, account, token, 0);

    // Lookup global state from UserManager
    UserManagerState memory userManagerState = _getUserManagerState(userManager);

    uint256 amount = _calculateRewardsByBlocks(
        account,
        token,
        pastBlocks,
        userInfo,
        userManagerState,
        userManagerAccountState
    );
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L710-L715

```
/**
 *  @dev collect staker rewards from the comptroller
 */
function withdrawRewards() external whenNotPaused nonReentrant {
    comptroller.withdrawRewards(msg.sender, stakingToken);
}
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L664-L667

```
function stake(uint96 amount) public whenNotPaused nonReentrant {
    IERC20Upgradeable erc20Token = IERC20Upgradeable(stakingToken);

    comptroller.withdrawRewards(msg.sender, stakingToken);
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L691-L698

```
function unstake(uint96 amount) external whenNotPaused nonReentrant {
    Staker storage staker = stakers[msg.sender];

    // Stakers can only unstaked stake balance that is unlocked. Stake balance
    // becomes locked when it is used to underwrite a borrow.
    if (staker.stakedAmount - staker.locked < amount) revert
↪   InsufficientBalance();
```

```
    comptroller.withdrawRewards(msg.sender, stakingToken);
```

I.e. there are no others means to actualize the rewards, the users can update whenever they want and this can be the only update for them.

This allows to remove both credit risk and interest expenses. I.e. there is no dilemma either lend to self and pay the interest or lend to someone else and bear the credit risk: it's possible to do nothing, wait for any desired time, then lend to self for 1 block, neither paying any meaningful interest, nor bearing any credit risk, and reap the UNION rewards for the whole period with the maximal multiplier.

Tool used

Manual Review

Recommendation

Member registration and trust utilization change, i.e. borrowings and repayments, should trigger reward accounting update to correctly reflect the timing of changes in the reward formula.

As the most direct mitigation consider adding the staker's reward update to `registerMember`, `updateLocked` and `debtWriteOff` functions that change the account state and the values of the corresponding reward formula components.

registerMember():

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L633-L654

```
    function registerMember(address newMember) public virtual whenNotPaused {
        if (stakers[newMember].isMember) revert NoExistingMember();

        uint256 count = 0;
        uint256 vouchersLength = vouchers[newMember].length;

        // Loop through all the vouchers to count how many active vouches there
        // are that are greater than 0. Vouch is the min of stake and trust
        for (uint256 i = 0; i < vouchersLength; i++) {
            Vouch memory vouch = vouchers[newMember][i];
            Staker memory staker = stakers[vouch.staker];
            if (staker.stakedAmount > 0) count++;
            if (count >= effectiveCount) break;
        }

        if (count < effectiveCount) revert NotEnoughStakers();
+   comptroller.withdrawRewards(newMember, stakingToken);
```

```
        stakers[newMember].isMember = true;
        IUnionToken(unionToken).burnFrom(msg.sender, newMemberFee);

        emit LogRegisterMember(msg.sender, newMember);
    }
```

updateLocked():

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L800-L841

```
    function updateLocked(
        address borrower,
        uint96 amount,
        bool lock
    ) external onlyMarket {
        uint96 remaining = amount;

        for (uint256 i = 0; i < vouchers[borrower].length; i++) {
            Vouch storage vouch = vouchers[borrower][i];
            uint96 innerAmount;

            if (lock) {
                // Look up the staker and determine how much unlock stake they
                // have available for the borrower to borrow. If there is 0
                // then continue to the next voucher in the array
                uint96 stakerLocked = stakers[vouch.staker].locked;
                uint96 stakerStakedAmount = stakers[vouch.staker].stakedAmount;
                uint96 availableStake = stakerStakedAmount - stakerLocked;
                uint96 lockAmount = _min(availableStake, vouch.trust -
↪   vouch.locked);
                if (lockAmount == 0) continue;

                // Calculate the amount to add to the lock then
                // add the extra amount to lock to the stakers locked amount
                // and also update the vouches locked amount and lastUpdated
↪   block
                innerAmount = _min(remaining, lockAmount);
+       comptroller.withdrawRewards(vouch.staker, stakingToken);
                stakers[vouch.staker].locked = stakerLocked + innerAmount;
                vouch.locked += innerAmount;
                vouch.lastUpdated = uint64(block.number);
            } else {
                // Look up how much this vouch has locked. If it is 0 then
                // continue to the next voucher. Then calculate the amount to
                // unlock which is the min of the vouches lock and what is
                // remaining to unlock
```

```
                uint96 locked = vouch.locked;
                if (locked == 0) continue;
                innerAmount = _min(locked, remaining);

                // Update the stored locked values and last updated block
+       comptroller.withdrawRewards(vouch.staker, stakingToken);
                stakers[vouch.staker].locked -= innerAmount;
                vouch.locked -= innerAmount;
                vouch.lastUpdated = uint64(block.number);
            }
```

debtWriteOff():

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L726-L755

```
    function debtWriteOff(
        address staker,
        address borrower,
        uint96 amount
    ) external {
        if (amount == 0) revert AmountZero();
        uint256 overdueBlocks = uToken.overdueBlocks();
        uint256 lastRepay = uToken.getLastRepay(borrower);

        // This function is only callable by the public if the loan is overdue by
        // overdue blocks + maxOverdueBlocks. This stops the system being left
↪   with
        // debt that is overdue indefinitely and no ability to do anything about
↪   it.
        if (block.number <= lastRepay + overdueBlocks + maxOverdueBlocks) {
            if (staker != msg.sender) revert AuthFailed();
        }

        Index memory index = voucherIndexes[borrower][staker];
        if (!index.isSet) revert VoucherNotFound();
        Vouch storage vouch = vouchers[borrower][index.idx];

        if (amount > vouch.locked) revert ExceedsLocked();

+   comptroller.withdrawRewards(staker, stakingToken);
        // update staker staked amount
        stakers[staker].stakedAmount -= amount;
        stakers[staker].locked -= amount;
        totalStaked -= amount;

        // update vouch trust amount
        vouch.trust -= amount;
```

```
        vouch.locked -= amount;
```


Issue M-1: Vouchers and vouchees indices become corrupted by UserManager's cancelVouch

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/157

Found by

Ch_301, hyh, yixxas, TurnipBoy, Picodes, obront, peanuts, Lambda

Summary

UserManager's cancelVouch() doesn't update `voucherIndexes` and `voucheeIndexes` entries for the last `vouchers` and `vouchees` arrays element that was moved to the position of the deleted element, keeping the non-existent link that can be filled with new data thereafter, making old element pointing to the new trust/locked values, that are incorrect for it.

Vulnerability Detail

Broken entries that is created this way has the immediate effect of unavailability of the several vouch operating functions for the corresponding staker and borrower. I.e. these functions obtain an index, which is not longer valid (index is the current `length` of the array), and revert on trying to access the corresponding element.

Moreover, if an additional voucher either for the borrower or staker be created with updateTrust(), two elements of the corresponding Indices array will point to the same new element, which has generally different locked/trust amount than the old one, whose pointer was previously lost. This way the old element will have incorrect trust/locked amount returned by these functions.

Impact

Immediate impact is unavailability of getLockedStake(), getVouchingAmount(), updateTrust(), cancelVouch() and debtWriteOff() for the borrower-staker combination that was this last element cancelVouch() moved.

Furthermore, if a new entry is placed, which is a high probability event being a part of normal activity, then old entry will point to incorrect staked/locked amount, and a various violations of the related constrains become possible.

Some examples are: trust can be updated via updateTrust() to be less then real locked amount; vouch cancellation can become blocked (say new voucher borrower gains a long-term lock and old lock with misplaced index cannot be cancelled until new lock be fully cleared, i.e. potentially for a long time).

The total impact is up to freezing the corresponding staker's funds as this opens up a way for various exploitations, for example a old entry's borrower can use the situation of unremovable trust and utilize this trust that staker would remove in a normal course of operations, locking extra funds of the staker this way.

The whole issue is a violation of the core UNION vouch accounting logic, with misplaced indices potentially piling up. Placing overall severity to be **high**.

Code Snippet

cancelVouch() do not update voucherIndexes[][] entry for the last voucher (sitting at `vouchers[borrower].length-1`) that was moved to the `idx` index of the deleted one:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L587-L598

```
// Remove borrower from vouchers array by moving the last item into the position
// of the index being removed and then poping the last item off the array
vouchers[borrower][voucherIndex.idx] =
↪  vouchers[borrower][vouchers[borrower].length - 1];
vouchers[borrower].pop();
delete voucherIndexes[borrower][staker];

// Remove borrower from vouchee array by moving the last item into the position
// of the index being removed and then poping the last item off the array
Index memory voucheeIndex = voucheeIndexes[borrower][staker];
vouchees[staker][voucheeIndex.idx] = vouchees[staker][vouchees[staker].length -
↪  1];
vouchees[staker].pop();
delete voucheeIndexes[borrower][staker];
```

In other words, `vouchers` and `vouchees` arrays do not perform full `voucherIndexes` and `voucheeIndexes` indices update, missing the update for the moved element. This leads to the discrepancies in `voucherIndexes` and `voucheeIndexes` data as the `voucherIndexes[borrower][staker]` entry corresponding to the last vocher becomes incorrect, pointing to the non-existing element at the position equal to the length of the array.

As a result once cancelVouch() be completed, the getLockedStake(), getVouchingAmount(), updateTrust(), cancelVouch() and debtWriteOff() become unavailable for these borrower and staker. For example, `voucherIndexes[borrower][vouchers[borrower][voucherIndex.idx].staker]` will point to `vouchers[borrower].length` and all `vouchers[borrower][index.idx]` operations will revert with `indexoutofbounds`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L476-L486

```
/**
```

```
 *  @dev Get staker locked stake for a borrower
 *  @param staker Staker address
 *  @param borrower Borrower address
 *  @return LockedStake
 */
function getLockedStake(address staker, address borrower) external view returns
↪  (uint256) {
    Index memory index = voucherIndexes[borrower][staker];
    if (!index.isSet) return 0;
    return vouchers[borrower][index.idx].locked;
}
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L488-L499

```
/**
 *  @dev Get vouching amount
 *  @param _staker Staker address
 *  @param borrower Borrower address
 */
function getVouchingAmount(address _staker, address borrower) external view
↪  returns (uint256) {
    Index memory index = voucherIndexes[borrower][_staker];
    Staker memory staker = stakers[_staker];
    if (!index.isSet) return 0;
    uint96 trustAmount = vouchers[borrower][index.idx].trust;
    return trustAmount < staker.stakedAmount ? trustAmount : staker.stakedAmount;
}
```

If another vouch be added to Bob, instead of unavailability situation it becomes the incorrect vouch info used in the functions above situation. I.e. the array element for that stale index will be present, but the data there be from the newest entry, i.e. it will be available, but incorrect for the old entry.

This way, for example, incorrect `locked` allows to update the `trust` to be less than real `vouch.locked`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L533-L539

```
Index memory index = voucherIndexes[borrower][staker];
if (index.isSet) {
    // Update existing record checking that the new trust amount is
    // not less than the amount of stake currently locked by the borrower
    Vouch storage vouch = vouchers[borrower][index.idx];
    if (trustAmount < vouch.locked) revert TrustAmountLtLocked();
    vouch.trust = trustAmount;
```

As another example using `locked>0` from the newest entry will prohibit the cancellation of the non-used old one:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L577-L585

```
function cancelVouch(address staker, address borrower) public
↪   onlyMember(msg.sender) whenNotPaused {
    if (staker != msg.sender && borrower != msg.sender) revert AuthFailed();

    Index memory voucherIndex = voucherIndexes[borrower][staker];
    if (!voucherIndex.isSet) revert VoucherNotFound();

    // Check that the locked amount for this vouch is 0
    Vouch memory vouch = vouchers[borrower][voucherIndex.idx];
    if (vouch.locked > 0) revert LockedStakeNonZero();
```

In this case borrower from the old entry can use this to utilized trust that meant to be cancelled by the staker, effectively freezing the corresponding funds.

Tool used

Manual Review

Recommendation

Consider adding the update for the elements moved:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L587-L598

```
        // Remove borrower from vouchers array by moving the last item into the
↪   position
        // of the index being removed and then poping the last item off the array
        vouchers[borrower][voucherIndex.idx] =
↪   vouchers[borrower][vouchers[borrower].length - 1];
+   voucherIndexes[borrower][vouchers[borrower][voucherIndex.idx].staker].idx =
↪   voucherIndex.idx;
        vouchers[borrower].pop();
        delete voucherIndexes[borrower][staker];

        // Remove borrower from vouchee array by moving the last item into the
↪   position
        // of the index being removed and then poping the last item off the array
        Index memory voucheeIndex = voucheeIndexes[borrower][staker];
        vouchees[staker][voucheeIndex.idx] =
↪   vouchees[staker][vouchees[staker].length - 1];
```

```
+   voucheeIndexes[vouchees[staker][voucheeIndex.idx].borrower][staker].idx =
↪   voucheeIndex.idx;
        vouchees[staker].pop();
    delete voucheeIndexes[borrower][staker];
```

Issue M-2: A stake that has just been locked gets full reward multiplier

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/142

Found by

Bahurum

Summary

A staker gets rewarded with full multiplier even if its stake has just been locked. Multiplier calculation should take into account the duration of the lock.

Vulnerability Detail

1. A staker stakes an amount of tokens.
2. The staker waits for some time
3. The staker has control of another member (bribe, ...)
4. The staker vouches this other member
5. The member borrows
6. The staker calls `Comptroller:withdrawRewards` and gets an amount of rewards with a multiplier corresponding to a locked stake
7. The member repays the loan

Note that steps 4 to 7 can be made in one tx, so no interest is paid at step 7.

The result is that the staker can always get the full multiplier for rewards, without ever putting any funds at risk, nor any interest being paid. This is done at the expense of other honest stakers, who get proprotionally less of the rewards dripped into the comptroller.

For a coded PoC replace the test `"stakerwithlockedbalancegetsmorerewards"` in `staking.ts` with the following

```
it("PoC: staker with locked balance gets more rewards even when just locked",
↪  async () => {
    const trustAmount = parseUnits("2000");
    const borrowAmount = parseUnits("1800");
    const [account, staker, borrower] = members;
```

```
    const [accountStaked, borrowerStaked, stakerStaked] = await
↪   helpers.getStakedAmounts(
        account,
        staker,
        borrower
    );

    expect(accountStaked).eq(borrowerStaked);
    expect(borrowerStaked).eq(stakerStaked);

    await helpers.updateTrust(staker, borrower, trustAmount);

    await roll(10);
    await helpers.borrow(borrower, borrowAmount); // borrows just after
↪   withdrawing

    const [accountMultiplier, stakerMultiplier] = await
↪   helpers.getRewardsMultipliers(account, staker);
    console.log("accountMultiplier: ", accountMultiplier);
    console.log("StakerMultiplier: ", stakerMultiplier);
    expect(accountMultiplier).lt(stakerMultiplier);  // the multiplier is larger
↪   even if just locked
});
```

Impact

A staker can get larger rewards designed for locked stakes by locking and unlocking in the same tx.

Code Snippet

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L166-L173

```
function getRewardsMultiplier(address account, address token) external view
↪   override returns (uint256) {
    IUserManager userManagerContract = _getUserManager(token);
    uint256 stakingAmount = userManagerContract.getStakerBalance(account);
    uint256 lockedStake = userManagerContract.getTotalLockedStake(account);
    (uint256 totalFrozen, ) = userManagerContract.getFrozenInfo(account,
↪   block.number);
    bool isMember = userManagerContract.checkIsMember(account);
    return _getRewardsMultiplier(stakingAmount, lockedStake, totalFrozen,
↪   isMember);
}
```

Tool used

Manual Review

Recommendation

Should introduce the accounting of the duration of a lock into the rewards calculation, so that full multiplier is given only to a lock that is as old as the stake itself.

Discussion

kingjacob

probably a medium because no one is losing funds, just some people might get more UNION votes.

Issue M-3: Removed adapter can still hold funds, removed token can still be deposited to a market

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/126

Found by

yixxas, Jeiwan

Summary

Removed adapter can still hold funds, removed token can still be deposited to a market.

Vulnerability Detail

When removing an adapter, there's no check for whether it's still holding funds (AssetManager.sol#L440). The same is true for tokens: when removing a token, there's no check for whether any of the supported adapters is still holding assets in this token.

Impact

In case an adapter that's being removed still holds funds, these funds will be removed from the total TVL until the removed adapter is re-added. And if there's no plans to re-add the adapter, the remaining funds will be locked in the adapter indefinitely. Similarly to tokens: if any of the supported markets is still holding a token that's being removed, the token assets held by the market will be removed from the total TVL and users won't be able to get their funds.

Code Snippet

AssetManager.sol#L440:

```
function removeAdapter(address adapterAddress) external override onlyAdmin {
    bool isExist = false;
    uint256 index;
    uint256 moneyMarketsLength = moneyMarkets.length;

    for (uint256 i = 0; i < moneyMarketsLength; i++) {
        if (adapterAddress == address(moneyMarkets[i])) {
            isExist = true;
            index = i;
            break;
        }
```

```
    }

    if (isExist) {
        moneyMarkets[index] = moneyMarkets[moneyMarketsLength - 1];
        moneyMarkets.pop();
    }
}
```

AssetManager.sol#L396:

```
function removeToken(address tokenAddress) external override onlyAdmin {
    bool isExist = false;
    uint256 index;
    uint256 supportedTokensLength = supportedTokensList.length;

    for (uint256 i = 0; i < supportedTokensLength; i++) {
        if (tokenAddress == address(supportedTokensList[i])) {
            isExist = true;
            index = i;
            break;
        }
    }

    if (isExist) {
        supportedTokensList[index] = supportedTokensList[supportedTokensLength -
↪   1];
        supportedTokensList.pop();
        supportedMarkets[tokenAddress] = false;
    }
}
```

Tool used

Manual Review

Recommendation

In the `removeAdapter` function, check market's supply by calling `moneyMarket.getSupply(token)` before removing an adapter. In the `removeToken` function, iterate over all supported adapters and check if they're still holding the token by calling `moneyMarket.getSupply(token)`.

Issue M-4: Stakers can have their funds locked for an extended period not related to the performance of their borrowers

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/114

Found by

hyh

Summary

Some stakers will have their funds lent locked for an extended period of time as partial prepayments to vouch logic (i.e. what vouch to prepay if a borrower provided the funds) depends on vouch order in the borrower's `vouchers` array, while cancelVouch() breaks up the FIFO vouch order it initially was build on.

Vulnerability Detail

While voucher array is initially ordered to favor the older lenders, i.e. in FIFO order, this initial order gets broken up over time as cancelVouch() places the very last lender (i.e. one who entered the latest) to the position of the removed one. This is a standard take on array element removal, which violates the redemption business logic in this case.

As a result stakers whose borrowers pay interest in full and do partial redemptions will have their funds locked with the lower priority in the prepayment queue just because some older vouch gets removed and a big vouch from the very end was moved before them.

Say Mike the lender was `2nd` lender for Alice the borrower, who borrowed more funds from various stakers over time, say `5` in total, and have Bob the big lender placed `5th` on her vouchers array as he entered the last.

Now Jade, Alice's `1st` lender, decided to remove the trust as she got payed back in full and the funds are needed elsewhere. `cancelVouch(Jade,Alice)` was called and Alice vouch array length is reduced to be `4`.

Bob gets placed `1st`, Mike is still `2nd`. Now suppose Mike lent Alice 1 year ago and it was `1kDAI`, while Bob lender yesterday and it was `10kDAI`. Now Alice prepayments will go towards Bob instead of Mike, who has his funds frozen until (and if) Bob's part be payed in full.

Impact

Net impact is temporal funds freeze if Alice remains to be in good health, and permanent fund freeze if Alice stops paying before Mike's turn of the redemptions. I.e. the prepayment order can determine if Mike loan end up being paid or not. Bob has his situation improved on Mike's behalf, who has worsened perspectives of the overall repayment.

This funds freeze is conditional on cancelVouch() calls, but as it needs to be run with ordinary parameters (i.e. remove **any** old vouch) and it is a typical operation (it will be run from time to time by stakers as their `vouchees` array has limited size), while core UNION logic of lender to borrower correspondence is broken here, so setting the severity to be **high**.

Code Snippet

cancelVouch() switches the vouch being removed with the last one:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L577-L591

```
function cancelVouch(address staker, address borrower) public
↪   onlyMember(msg.sender) whenNotPaused {
    if (staker != msg.sender && borrower != msg.sender) revert AuthFailed();

    Index memory voucherIndex = voucherIndexes[borrower][staker];
    if (!voucherIndex.isSet) revert VoucherNotFound();

    // Check that the locked amount for this vouch is 0
    Vouch memory vouch = vouchers[borrower][voucherIndex.idx];
    if (vouch.locked > 0) revert LockedStakeNonZero();

    // Remove borrower from vouchers array by moving the last item into the
↪   position
    // of the index being removed and then poping the last item off the array
    vouchers[borrower][voucherIndex.idx] =
↪   vouchers[borrower][vouchers[borrower].length - 1];
    vouchers[borrower].pop();
    delete voucherIndexes[borrower][staker];
```

This logic is the classic take on array removal, but `vouchers[borrower]` array order is material, being used in updateLocked() assuming that array index is a good proxy for loan age, and adhering to the first in, first out logic:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L800-L841

```
function updateLocked(
```

```
    address borrower,
    uint96 amount,
    bool lock
) external onlyMarket {
    uint96 remaining = amount;

    for (uint256 i = 0; i < vouchers[borrower].length; i++) {
        Vouch storage vouch = vouchers[borrower][i];
        uint96 innerAmount;

        if (lock) {
            // Look up the staker and determine how much unlock stake they
            // have available for the borrower to borrow. If there is 0
            // then continue to the next voucher in the array
            uint96 stakerLocked = stakers[vouch.staker].locked;
            uint96 stakerStakedAmount = stakers[vouch.staker].stakedAmount;
            uint96 availableStake = stakerStakedAmount - stakerLocked;
            uint96 lockAmount = _min(availableStake, vouch.trust - vouch.locked);
            if (lockAmount == 0) continue;

            // Calculate the amount to add to the lock then
            // add the extra amount to lock to the stakers locked amount
            // and also update the vouches locked amount and lastUpdated block
            innerAmount = _min(remaining, lockAmount);
            stakers[vouch.staker].locked = stakerLocked + innerAmount;
            vouch.locked += innerAmount;
            vouch.lastUpdated = uint64(block.number);
        } else {
            // Look up how much this vouch has locked. If it is 0 then
            // continue to the next voucher. Then calculate the amount to
            // unlock which is the min of the vouches lock and what is
            // remaining to unlock
            uint96 locked = vouch.locked;
            if (locked == 0) continue;
            innerAmount = _min(locked, remaining);

            // Update the stored locked values and last updated block
            stakers[vouch.staker].locked -= innerAmount;
            vouch.locked -= innerAmount;
            vouch.lastUpdated = uint64(block.number);
        }
```

updateLocked() is called when a borrower repays current debt interest in full:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L573-L619

```
function _repayBorrowFresh(
```


```
    address payer,
    address borrower,
    uint256 amount
) internal {
    if (!accrueInterest()) revert AccrueInterestFailed();

    uint256 interest = calculatingInterest(borrower);
    uint256 borrowedAmount = borrowBalanceStoredInternal(borrower);
    uint256 repayAmount = amount > borrowedAmount ? borrowedAmount : amount;
    if (repayAmount == 0) revert AmountZero();

    uint256 toReserveAmount;
    uint256 toRedeemableAmount;

    if (repayAmount >= interest) {
        ...

        // Call update locked on the userManager to lock this borrowers stakers.
↪   This function
        // will revert if the account does not have enough vouchers to cover the
↪   repay amount. ie
        // the borrower is trying to repay more than is locked (owed)
        IUserManager(userManager).updateLocked(borrower, uint96(repayAmount -
↪   interest), false);
```

Which vouch to be updated by updateLocked(), i.e. where to allocate this new repayment if a borrower has many locked vouches, is material as it determines who gets the money back:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L432-L466

```
/**
 *  @dev Get frozen coin age
 *  @param  staker Address of staker
 *  @param  pastBlocks Number of blocks past to calculate coin age from
 *          coin age = min(block.number - lastUpdated, pastBlocks) * amount
 */
function getFrozenInfo(address staker, uint256 pastBlocks)
    public
    view
    returns (uint256 memberTotalFrozen, uint256 memberFrozenCoinAge)
{
    uint256 overdueBlocks = uToken.overdueBlocks();
    uint256 voucheesLength = vouchees[staker].length;
    // Loop through all of the stakers vouchees sum their total
    // locked balance and sum their total memberFrozenCoinAge
    for (uint256 i = 0; i < voucheesLength; i++) {
```

```
        // Get the vouchee record and look up the borrowers voucher record
        // to get the locked amount and lastUpdate block number
        Vouchee memory vouchee = vouchees[staker][i];
        Vouch memory vouch = vouchers[vouchee.borrower][vouchee.voucherIndex];

        uint256 lastUpdated = vouch.lastUpdated;
        uint256 diff = block.number - lastUpdated;

        if (overdueBlocks < diff) {
            uint96 locked = vouch.locked;
            memberTotalFrozen += locked;
            if (pastBlocks >= diff) {
                memberFrozenCoinAge += (locked * diff);
            } else {
                memberFrozenCoinAge += (locked * pastBlocks);
            }
        }
    }
}
```

This can mean who gets the money faster or who gets the money at all, depending on the future behavior of the borrower.

Tool used

Manual Review

Recommendation

A simplest solution is to do it in a hard way and cycle through the whole array, for example:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L577-L591

```
    function cancelVouch(address staker, address borrower) public
↪   onlyMember(msg.sender) whenNotPaused {
        if (staker != msg.sender && borrower != msg.sender) revert AuthFailed();

        Index memory voucherIndex = voucherIndexes[borrower][staker];
        if (!voucherIndex.isSet) revert VoucherNotFound();

        // Check that the locked amount for this vouch is 0
        Vouch memory vouch = vouchers[borrower][voucherIndex.idx];
        if (vouch.locked > 0) revert LockedStakeNonZero();
```

```
        // Remove borrower from vouchers array by moving the last item into the
↪   position
        // of the index being removed and then poping the last item off the array
+       ... // create new empty array
+       for (uint256 i = 0; i < vouchers[borrower].length; i++) {
+           Vouch storage vouch = vouchers[borrower][i];
+           ... // if `i <> voucherIndex.idx` and vouch has positive trust put it
↪   to the new array
+       for (uint256 i = 0; i < vouchers[borrower].length; i++) {
+           Vouch storage vouch = vouchers[borrower][i];
+       ... // if `i <> voucherIndex.idx` and vouch has zero trust put it to the
↪   new array, keeping the old order among them this way
+       ... // replace old `vouches` with new array
-       vouchers[borrower][voucherIndex.idx] =
↪   vouchers[borrower][vouchers[borrower].length - 1];
-       vouchers[borrower].pop();

        delete voucherIndexes[borrower][staker];
```

It can be feasible as `vouchers` length will not be big in the most cases, cancelVouch() isn't that frequent, and loan order is material for business logic to bear additional gas costs.

Also, this reorganisation of non-zero vouches first, zeros later eliminates the potential issue with stakers leaving vouchers with zero amounts just to keep the place in the redemption queue, which also doesn't adhere to FIFO logic.

Discussion

kingjacob

This is as designed and a function of fifo.

dmitriia

The issue is that `cancelVouch` as it is **breaks** FIFO.

As a result, the lenders can be compensated randomly, not according to FIFO.

kingjacob

If cancel vouch changes the locked status of stakers of an outstanding borrow without a borrow or repay happening that would be very bad.

But if im understanding the report as borrower locks 1,2,3. Repays enough to unlock 1. 1 cancels. The order is now 3,2. That doesnt significantly change the risk for 2. The risk is still that the borrower you vouched doesnt repay. Theres maybe a marginal risk, But practically its hard to imagine this scenario occurring separate from a straight default or long repayment cycle.

That said strict FIFO would be preferred for ease of explanation but looping through changes the cost from 1 to n, with each n costing ~5000 gas per N. So its a tradeoff between this edgecase vs more people being able to afford to cancel bad or stale vouches.

dmitriia

Yes, when the oldest vouch 1 is cancelled, the newest one 3 becomes the oldest, while the previous one, 2, keeps its place.

The implications of this isn't just a slight disturbing of the FIFO order, the resulting order can become fully random. This can be a material consideration for lenders and honest borrowers.

Say Bob lends to Amelie through the system, while she is a big borrower with lots of connections, i.e. she is a vouchee for many vouchers, and use this trust a lot.

Now Bob wants to have his money back and asks Amelie, who isn't insolvent and is able and willing to return the debt to Bob. But she can't as the system with repay the debts in nearly random order originated from the sequence of vouches cancellations and new vouches introductions.

As she can't return all debt to all the borrowers or any substantial part of them at this point, it is simply too much liquidity at once, this queue can't be cleared. Bob can't be repaid due to mere technical issue.

This means that the term of a loan cannot be deemed fixed even if both parties agree on it and do honour this agreement.

But rates do differ drastically depending on the term of a loan. This is the `yieldcurve` and it is steep most of the times. Current interest rate Union employs can be good for say 3 months loan, but insufficient for 3 years one.

This rate can't be just set higher as it is the matter of credit spread, the difference between this rate and risk-free rate for the *term* of a loan. Risk free rates for different terms are quite different, per risk free yield curve.

For example, if the rate is set to be high it will mean sufficient spread for longer terms, but overly big for shorter terms and honest borrowers will be reluctant to get loans, and vice versa.

That means keeping FIFO intact is crucial and very well justifies gas costs. Those can be kept lower via, for example, additional array `voucherOrder[borrower]` with historical order of the vouchers and accessing `vouchers` across the code not as `vouchers[borrower][i],i=0,1,...`, but as `vouchers[borrower][voucherOrder[borrower][i]],i=0,1,...`.

The code becomes:

```
    function cancelVouch(address staker, address borrower) public
↪   onlyMember(msg.sender) whenNotPaused {
```

```
        if (staker != msg.sender && borrower != msg.sender) revert AuthFailed();

        Index memory voucherIndex = voucherIndexes[borrower][staker];
        if (!voucherIndex.isSet) revert VoucherNotFound();

        // Check that the locked amount for this vouch is 0
        Vouch memory vouch = vouchers[borrower][voucherIndex.idx];
        if (vouch.locked > 0) revert LockedStakeNonZero();

+       bool memory reached;
+       for (uint16 i = 0; i < voucherOrder[borrower].length - 1; i++) {
+           if (!reached) {
+               if (voucherOrder[borrower][i] == voucherIndex.idx) reached =
↪   true;
+           }
+           if (reached) voucherOrder[borrower][i] = voucherOrder[borrower][i +
↪   1]
+       }
+       voucherOrder[borrower].pop();

        // Remove borrower from vouchers array by moving the last item into the
↪   position
        // of the index being removed and then poping the last item off the array
        vouchers[borrower][voucherIndex.idx] =
↪   vouchers[borrower][vouchers[borrower].length - 1];
        vouchers[borrower].pop();
```

`voucherOrder.push(vouchers[borrower].length)` needs to be done before pushing to `vouchers[borrower]` in updateTrust().

Evert0x

Downgrading to medium severity after discussions with senior, protocol and internally.

Issue M-5: updateTrust() vouchers also need check maxVouchers

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/109

Found by

bin2chen

Summary

maxVouchers is to prevent the “vouchees“ array from getting too big and the loop will have the GAS explosion problem, but “vouchers“have the same problem, if you don't check the vouchers array, it is also possible that vouchers are big and cause updateLocked() to fail

Vulnerability Detail

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L548

vouchees check < maxVouchers ,but vouchers don't check

```
    function updateTrust(address borrower, uint96 trustAmount) external
    ↪  onlyMember(msg.sender) whenNotPaused {
...
            uint256 voucheesLength = vouchees[staker].length;
            if (voucheesLength >= maxVouchers) revert MaxVouchees();


            uint256 voucherIndex = vouchers[borrower].length;
            voucherIndexes[borrower][staker] = Index(true,
            ↪  uint128(voucherIndex));
            vouchers[borrower].push(Vouch(staker, trustAmount, 0, 0)); /****
            ↪  don't check maxVouchers****/
```

Impact

it is also possible that vouchers are big and cause updateLocked() to fail

Code Snippet

Tool used

Manual Review

Recommendation

```
   function updateTrust(address borrower, uint96 trustAmount) external
↪   onlyMember(msg.sender) whenNotPaused {
...
            uint256 voucheesLength = vouchees[staker].length;
            if (voucheesLength >= maxVouchers) revert MaxVouchees();


            uint256 voucherIndex = vouchers[borrower].length;
+         if (voucherIndex >= maxVouchers) revert MaxVouchees();
            voucherIndexes[borrower][staker] = Index(true,
↪   uint128(voucherIndex));
            vouchers[borrower].push(Vouch(staker, trustAmount, 0, 0));
```

Issue M-6: Unsafe downcasting arithmetic operation in UserManager related contract and in UToken.sol

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/96

Found by

8olidity, ctf_sec, Lambda

Summary

The value is unsafely downcasted and truncated from uint256 to uint96 or uint128 in UserManager related contract and in UToken.sol.

Vulnerability Detail

value can unsafely downcasted. let us look at it cast by cast.

In UserManagerDAI.sol

```
function stakeWithPermit(
    uint256 amount,
    uint256 nonce,
    uint256 expiry,
    uint8 v,
    bytes32 r,
    bytes32 s
) external whenNotPaused {
    IDai erc20Token = IDai(stakingToken);
    erc20Token.permit(msg.sender, address(this), nonce, expiry, true, v, r, s);

    stake(uint96(amount));
}
```

as we can see, the user's staking amount is downcasted from uint256 to uint96.

the same issue exists in UserManagerERC20.sol

In the context of UToken.sol, a bigger issue comes.

User invokes the borrow function in UToken.sol

```
function borrow(address to, uint256 amount) external override
↪   onlyMember(msg.sender) whenNotPaused nonReentrant {
```

and

```
// Withdraw the borrowed amount of tokens from the assetManager and send them to
↪  the borrower
if (!assetManagerContract.withdraw(underlying, to, amount)) revert
↪  WithdrawFailed();

// Call update locked on the userManager to lock this borrowers stakers. This
↪  function
// will revert if the account does not have enough vouchers to cover the borrow
↪  amount. ie
// the borrower is trying to borrow more than is able to be underwritten
IUserManager(userManager).updateLocked(msg.sender, uint96(amount + fee), true);
```

note when we withdraw fund from asset Manager, we use a uint256 amount, but we downcast it to uint96(amount + fee) when updating the locked. The accounting would be so broken if the amount + fee is a larger than uint96 number.

Same issue in the function UToken.sol# _repayBorrowFresh

```
function _repayBorrowFresh(
    address payer,
    address borrower,
    uint256 amount
) internal {
```

and

```
// Update the account borrows to reflect the repayment
accountBorrows[borrower].principal = borrowedAmount - repayAmount;
accountBorrows[borrower].interest = 0;
```

and

we use a uint256 number for borrowedAmount - repayAmount, but downcast it to uint96(repayAmount - interest) when updating the lock!

Note there are index-related downcasting, the damage is small , comparing the accounting related downcasting.because it is difference to have uint128 amount of vouch, but I still want to mention it: the index is unsafely downcasted from uint256 to uint128

```
// Get the new index that this vouch is going to be inserted at
// Then update the voucher indexes for this borrower as well as
// Adding the Vouch the the vouchers array for this staker
uint256 voucherIndex = vouchers[borrower].length;
voucherIndexes[borrower][staker] = Index(true, uint128(voucherIndex));
vouchers[borrower].push(Vouch(staker, trustAmount, 0, 0));
```

```
// Add the voucherIndex of this new vouch to the vouchees array for this
// staker then update the voucheeIndexes with the voucheeIndex
uint256 voucheeIndex = voucheesLength;
vouchees[staker].push(Vouchee(borrower, uint96(voucherIndex)));
voucheeIndexes[borrower][staker] = Index(true, uint128(voucheeIndex));
```

There are block.number related downcasting, which is a smaller issue.

```
vouch.lastUpdated = uint64(block.number);
```

Impact

The damage level from the number truncation is rated by:

UToken borrow and repaying downcasting > staking amount downcating truncation > the vouch index related downcasting. > block.number casting.

Code Snippet

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L550-L563

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManagerDAI.sol#L28

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManagerERC20.sol#L26

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L552

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L619

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L827

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L840

Tool used

Manual Review

Recommendation

Just use uint256, or use openzepplin safeCasting.

https://docs.openzeppelin.com/contracts/3.x/api/utils#SafeCast

Discussion

kingjacob

will likely fix with a safecast lib

Issue M-7: UserManager.sol#L438-L466 : getFrozenInfo could revert due to out of gas when the vouchees array size is large

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/87

Found by

ak1

Summary

The function `getFrozenInfo` travers the vouchees array and calculates `uint256memberTotalFrozen,uint256memberFrozenCoinAge`. When the vouchees array is large, calling this function could revert due to out of gas.

Vulnerability Detail

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L438-L466

The function `getFrozenInfo` travers the vouchees array and calculates `uint256memberTotalFrozen,uint256memberFrozenCoinAge`. When the vouchees array is large, calling this function could revert due to out of gas.

Loading of two type of struct data in memory and math operations are done, this could be gas costly and could revert when vouchee size is large.

Impact

This affect the functions wherever the getFrozenInfo is called. https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L863 https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L881

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L889

Code Snippet

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L438-L466

Tool used

Manual Review

Recommendation

Put cap on number of vouchee size. Do operation based on upto certain index and then do for others. Do not use for full length of array in single shot.

Discussion

dmitriia

`vouchees` do have cap on the length: `if(voucheesLength>=maxVouchers)revertMaxVouchees()` in updateTrust(). Name still needs to be corrected though to be `maxVouchees`.

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L548

Anyway it's fully valid for the index introduction part.

Issue M-8: getUserInfo() returns incorrect values for locked and stakedAmount

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/80

Found by

obront

Summary

The `getUserInfo()` function mixes up the values for `locked` and `stakedAmount`, so the value for each of these is returned for the other.

Vulnerability Detail

In UnionLens.sol, the `getUserInfo()` function is used to retrieve information about a given user.

In order to pull the user's staking information, the following function is called:

```
(bool isMember, uint96 locked, uint96 stakedAmount) = userManager.stakers(user);
```

This function is intended to return these three values from the UserManager.sol contract. However, in that contract, the function being called returns a Staker struct, which has the following values:

```
struct Staker {
    bool isMember;
    uint96 stakedAmount;
    uint96 locked;
}
```

Because both `locked` and `stakedAmount` have the type `uint96`, the function does not revert, and simply returns the incorrect values to the caller.

Impact

Any user or front end calling the `getUserInfo()` function will be given incorrect values, which could lead to wrong decisions.

Code Snippet

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/UnionLens.sol#L62

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L37-L41

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L117-L120

Tool used

Manual Review

Recommendation

Reverse the order of return values in the `getUserInfo()` function, so that it reads:

```
(bool isMember, uint96 stakedAmount, uint96 locked) = userManager.stakers(user);
```

Issue M-9: Priority withdrawal sequence array will grow infinitely over time

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/76

Found by

hyh, hansfriese, yixxas, GimelSec, Lambda

Summary

Withdraw sequence reduction operation isn't present on adapter removal, while new withdraw sequence is required to have the same length as the old one if set directly. I.e. this withdraw sequence length is kept as current invariant, which only increases on each adapter addition, this way growing indefinitely over time.

Vulnerability Detail

Each `removeAdapter()->addAdapter()` operation sequence makes `withdrawSeq` array to be `1` item longer, while `moneyMarkets` array preserves its length. As `withdrawSeq` is just an ordering of `moneyMarkets`, this is not desirable, but cannot be directly fixed as administrative setWithdrawSequence() requires the length to be preserved, while additional addAdapter() operations keep the length difference.

As an example, suppose the system operates long enough and now there are `5` adapters, while removeAdapter() was run 50 times since the contract deployment. `withdrawSeq` will have length of `55` and it is impossible to reduce it.

Impact

All the users end up paying increased gas costs as AssetManager's withdraw() iterates over the full `withdrawSeq` array.

Over time the length of this array will increase to make withdraw() requiring too much gas, making the operation oftentimes economically non-viable. This is temporal funds freeze, i.e. a rational user will have to delay the withdrawal up to the moment when bloated gas cost will become justified, not being able to withdraw funds without an additional loss bigger than an acceptable threshold until then.

With the further growth of the `withdrawSeq`, the block gas limit can be surpassed, making the operation overall forbidden. This is permanent fund freeze impact conditional on system being in production long enough.

Code Snippet

setWithdrawSequence() requires new sequence array to be the same size as the current `withdrawSeq`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L135-L142

```
/**
 *  @dev Set withdraw sequence
 *  @param newSeq priority sequence of money market indices to be used while
 ↪  withdrawing
 */
function setWithdrawSequence(uint256[] calldata newSeq) external override
 ↪  onlyAdmin {
    if (newSeq.length != withdrawSeq.length) revert NotParity();
    withdrawSeq = newSeq;
}
```

removeAdapter() reduces the length of the `moneyMarkets` array, but not of the `withdrawSeq` array:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L436-L457

```
/**
 *  @dev Remove a adapter for the underlying lending protocol
 *  @param adapterAddress adapter address
 */
function removeAdapter(address adapterAddress) external override onlyAdmin {
    bool isExist = false;
    uint256 index;
    uint256 moneyMarketsLength = moneyMarkets.length;

    for (uint256 i = 0; i < moneyMarketsLength; i++) {
        if (adapterAddress == address(moneyMarkets[i])) {
            isExist = true;
            index = i;
            break;
        }
    }

    if (isExist) {
        moneyMarkets[index] = moneyMarkets[moneyMarketsLength - 1];
        moneyMarkets.pop();
    }
}
```

In the same time addAdapter() increases both `moneyMarkets` and `withdrawSeq` arrays by one when being called with `adapterAddress` that's not already present:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L416-L434

```
/**
 *  @dev Add a new adapter for the underlying lending protocol
 *  @param adapterAddress adapter address
 */
function addAdapter(address adapterAddress) external override onlyAdmin {
    bool isExist = false;
    uint256 moneyMarketsLength = moneyMarkets.length;

    for (uint256 i = 0; i < moneyMarketsLength; i++) {
        if (adapterAddress == address(moneyMarkets[i])) isExist = true;
    }

    if (!isExist) {
        moneyMarkets.push(IMoneyMarketAdapter(adapterAddress));
        withdrawSeq.push(moneyMarkets.length - 1);
    }

    approveAllTokensMax(adapterAddress);
}
```

withdraw() iterates over the full `withdrawSeq` array each time:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L345-L359

```
if (isMarketSupported(token)) {
    uint256 withdrawSeqLength = withdrawSeq.length;
    // iterate markets according to defined sequence and withdraw
    for (uint256 i = 0; i < withdrawSeqLength && remaining > 0; i++) {
        IMoneyMarketAdapter moneyMarket = moneyMarkets[withdrawSeq[i]];
        if (!moneyMarket.supportsToken(token)) continue;

        uint256 supply = moneyMarket.getSupply(token);
        if (supply == 0) continue;

        uint256 withdrawAmount = supply < remaining ? supply : remaining;
        remaining -= withdrawAmount;
        moneyMarket.withdraw(token, account, withdrawAmount);
    }
}
```

Tool used

Manual Review

Recommendation

Consider updating the setWithdrawSequence() logic to require new sequence array to be the same size as `moneyMarkets` whose ordering it represents:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L135-L142

```
    /**
     *  @dev Set withdraw sequence
     *  @param newSeq priority sequence of money market indices to be used while
↪   withdrawing
     */
    function setWithdrawSequence(uint256[] calldata newSeq) external override
↪   onlyAdmin {
-       if (newSeq.length != withdrawSeq.length) revert NotParity();
+       if (newSeq.length != moneyMarkets.length) revert NotParity();
        withdrawSeq = newSeq;
    }
```

Issue M-10: `AssetManager.rebalance()` will revert when the balance of `tokenAddress` in the money market is 0.

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/75

Found by

hansfriese, ctf_sec

Summary

`AssetManager.rebalance()` will revert when the balance of `tokenAddress` in the money market is 0.

Vulnerability Detail

AssetManager.rebalance() tries to withdraw tokens from each money market for re-balancing here.

```
// Loop through each money market and withdraw all the tokens
for (uint256 i = 0; i < moneyMarketsLength; i++) {
    IMoneyMarketAdapter moneyMarket = moneyMarkets[i];
    if (!moneyMarket.supportsToken(tokenAddress)) continue;
    moneyMarket.withdrawAll(tokenAddress, address(this));

    supportedMoneyMarkets[supportedMoneyMarketsSize] = moneyMarket;
    supportedMoneyMarketsSize++;
}
```

When the balance of the `tokenAddress` is 0, we don't need to call `moneyMarket.withdrawAll()` but it still tries to call.

But this will revert because Aave V3 doesn't allow to withdraw 0 amount here.

```
function validateWithdraw(
  DataTypes.ReserveCache memory reserveCache,
  uint256 amount,
  uint256 userBalance
) internal pure {
  require(amount != 0, Errors.INVALID_AMOUNT);
```

So `AssetManager.rebalance()` will revert if one money market has zero balance of `tokenAddress`.

Impact

The money markets can't be rebalanced if there is no balance in at least one market.

Code Snippet

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L514

Tool used

Manual Review

Recommendation

I think we can modify AaveV3Adapter.withdrawAll() to work only when the balance is positive.

```
function withdrawAll(address tokenAddress, address recipient)
    external
    override
    onlyAssetManager
    checkTokenSupported(tokenAddress)
{
    address aTokenAddress = tokenToAToken[tokenAddress];
    IERC20Upgradeable aToken = IERC20Upgradeable(aTokenAddress);
    uint256 balance = aToken.balanceOf(address(this));

    if (balance > 0) {
        lendingPool.withdraw(tokenAddress, type(uint256).max, recipient);
    }
}
```

Issue M-11: gas limit DoS via unbounded operations

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/69

Found by

Ch_301, ctf_sec

Summary

Only one attack will lead to two types of vulnerabilities in `UserManager.sol` and `UToken.sol`

Vulnerability Detail

On `UserManager.sol ==> updateTrust()` Case one: malicious users (members) can keep `vouching` Alice with `trustAmount==0` until his `vouchers` array achieves the max limit (2**256-1) So when a normal member tries to give `vouching` to Alice with `trustAmount!=0` he will find because the `vouchers` array completely full.

Case two (which is more realistic): malicious users (members) can keep `vouching` Alice with `trustAmount==0` until his `vouchers` array achieves late's say 20% of max limit (2**256-1) The problem is when Alice invoke `borrow()` or `repayBorrow()` on `UToken.sol`

```
 IUserManager(userManager).updateLocked(msg.sender, uint96(amount + fee), true);
...
IUserManager(userManager).updateLocked(borrower, uint96(repayAmount - interest),
↪  false);
```

It will call `updateLocked()` on `UserManager.sol`

```
function updateLocked(
     address borrower,
     uint96 amount,
     bool lock
 ) external onlyMarket {
     uint96 remaining = amount;

     for (uint256 i = 0; i < vouchers[borrower].length; i++) {
```

The for loop could go through `vouchers[]` which could be long enough to lead to a "gas limit DoS via unbounded operations" And the same thing with `registerMember()`, any user could lose all their fund in this transaction

```
function registerMember(address newMember) public virtual whenNotPaused {
 if (stakers[newMember].isMember) revert NoExistingMember();

 uint256 count = 0;
 uint256 vouchersLength = vouchers[newMember].length;

 // Loop through all the vouchers to count how many active vouches there
 // are that are greater than 0. Vouch is the min of stake and trust
 for (uint256 i = 0; i < vouchersLength; i++) {
```

Impact

1- The user couldn't get any more `vouching` 2- The user will be not able to `borrow()` or `repayBorrow()` 3- No one can in invoke`registerMember()` successfully for a specific user

Code Snippet

```
vouchers[borrower].push(Vouch(staker, trustAmount, 0, 0));
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L555

```
for (uint256 i = 0; i < vouchers[borrower].length; i++) {
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L807

```
   uint256 vouchersLength = vouchers[newMember].length;

// Loop through all the vouchers to count how many active vouches there
// are that are greater than 0. Vouch is the min of stake and trust
for (uint256 i = 0; i < vouchersLength; i++) {
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L637-L641

Tool used

Manual Review

Recommendation

Add check for `trustAmount==0`

Discussion

dmitriia

Some minimal `trustAmount` looks to be needed here as the same can be repeated with dust amounts (say 1 wei, as the attacker pays gas anyway, so financially it will not matter).

Issue M-12: Partial withdrawals by AssetManager lead to user funds freeze

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/50

Found by

hyh, Jeiwan

Summary

AssetManager's withdraw() doesn't guarantee the retrieval of the full requested amount. However, all dependant function always treat such withdrawal call as if full amount was successfully sent to a recipient.

Vulnerability Detail

Partial withdrawals by AssetManager are unaccounted in all withdraw initiating functions: user-facing UserManager's unstake(), UToken's borrow() and redeem(). This way the `remaining` amount withdraw() failed to obtain from the adapters is permanently lost for the withdrawal recipient as full amount is accounted each time.

Strategies AssetManager utilize via adapters can have temporal funds unavailability, say Aave and Compound can have liquidity squeezes. If a particular lending pool has liquidity shortage, i.e. almost all underlying is lent out, full withdrawal of the requested underlying token amount will not be possible at the moment. It doesn't mean the funds are lost, so writing the full amount off for a recipient isn't correct and is equivalent to user's fund freeze in the system.

Impact

Net impact is permanent fund freeze for the users who were recipients for such withdrawals. I.e. when `remaining` funds become available later, there is no way to receive them for the users as their accounting was updated by full amounts already. This way such remaining funds were de facto wrote down for the users and became excess unallocated funds of a Strategy, i.e. profit for the system from AssetManager's funds allocation.

As this permanent fund freeze is conditional on Strategy liquidity squeeze, which is medium probability event, being a part of normal activity for lending protocols, setting the severity to be **medium**.

Code Snippet

Accounting discrepancy is introduced each time as AssetManager's withdraw() do not guarantee full `amount` retrieval, always returns true and reduces the balances by `amount-remaining`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L328-L369

```
function withdraw(
    address token,
    address account,
    uint256 amount
) external override whenNotPaused nonReentrant onlyAuth(token) returns (bool) {
    if (!_checkSenderBalance(msg.sender, token, amount)) revert
↪   InsufficientBalance();

    uint256 remaining = amount;

    // If there are tokens in Asset Manager then transfer them on priority
    uint256 selfBalance = IERC20Upgradeable(token).balanceOf(address(this));
    if (selfBalance > 0) {
        uint256 withdrawAmount = selfBalance < remaining ? selfBalance :
↪   remaining;
        remaining -= withdrawAmount;
        IERC20Upgradeable(token).safeTransfer(account, withdrawAmount);
    }

    if (isMarketSupported(token)) {
        uint256 withdrawSeqLength = withdrawSeq.length;
        // iterate markets according to defined sequence and withdraw
        for (uint256 i = 0; i < withdrawSeqLength && remaining > 0; i++) {
            IMoneyMarketAdapter moneyMarket = moneyMarkets[withdrawSeq[i]];
            if (!moneyMarket.supportsToken(token)) continue;

            uint256 supply = moneyMarket.getSupply(token);
            if (supply == 0) continue;

            uint256 withdrawAmount = supply < remaining ? supply : remaining;
            remaining -= withdrawAmount;
            moneyMarket.withdraw(token, account, withdrawAmount);
        }
    }

    if (!_isUToken(msg.sender, token)) {
        balances[msg.sender][token] = balances[msg.sender][token] - amount +
↪   remaining;
        totalPrincipal[token] = totalPrincipal[token] - amount + remaining;
```

```
    }

    emit LogWithdraw(token, account, amount, remaining);

    return true;
}
```

The functions that use withdraw() treat it differently, always assuming that the whole amount is successfully retrieved.

I.e. UserManager's balance in AssetManager will be reduced less than user's balance in UserManager as UserManager's unstake() always removes the full `amount` from `staker.stakedAmount` and `totalStaked`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L691-L705

```
function unstake(uint96 amount) external whenNotPaused nonReentrant {
    Staker storage staker = stakers[msg.sender];

    // Stakers can only unstaked stake balance that is unlocked. Stake balance
    // becomes locked when it is used to underwrite a borrow.
    if (staker.stakedAmount - staker.locked < amount) revert
↪   InsufficientBalance();

    comptroller.withdrawRewards(msg.sender, stakingToken);

    staker.stakedAmount -= amount;
    totalStaked -= amount;

    if (!IAssetManager(assetManager).withdraw(stakingToken, msg.sender, amount))
↪   {
        revert AssetManagerWithdrawFailed();
    }
```

UToken's borrow() also always assumes that full `amount` is retrieved to the borrower, adding full `amount` to `accountBorrows` entry:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L512-L555

```
function borrow(address to, uint256 amount) external override
↪   onlyMember(msg.sender) whenNotPaused nonReentrant {
    IAssetManager assetManagerContract = IAssetManager(assetManager);
    if (amount < minBorrow) revert AmountLessMinBorrow();
    if (amount > getRemainingDebtCeiling()) revert AmountExceedGlobalMax();

    ...
```

```
    uint256 accountBorrowsNew = borrowedAmount + amount + fee;
    uint256 totalBorrowsNew = totalBorrows + amount + fee;

    // Update internal balances
    accountBorrows[msg.sender].principal += amount + fee;

    ...

    // Withdraw the borrowed amount of tokens from the assetManager and send them
↪   to the borrower
    if (!assetManagerContract.withdraw(underlying, to, amount)) revert
↪   WithdrawFailed();

    // Call update locked on the userManager to lock this borrowers stakers. This
↪   function
    // will revert if the account does not have enough vouchers to cover the
↪   borrow amount. ie
    // the borrower is trying to borrow more than is able to be underwritten
    IUserManager(userManager).updateLocked(msg.sender, uint96(amount + fee),
↪   true);

    emit LogBorrow(msg.sender, to, amount, fee);
}
```

UToken's redeem() similarly always burns full `uTokenAmount` corresponding to `underlyingAmount` requested from AssetManager:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L707-L740

```
function redeem(uint256 amountIn, uint256 amountOut) external override
↪   whenNotPaused nonReentrant {
    if (!accrueInterest()) revert AccrueInterestFailed();
    if (amountIn != 0 && amountOut != 0) revert AmountZero();

    uint256 exchangeRate = exchangeRateStored();

    // Amount of the uToken to burn
    uint256 uTokenAmount;

    // Amount of the underlying token to redeem
    uint256 underlyingAmount;

    if (amountIn > 0) {
        // We calculate the exchange rate and the amount of underlying to be
↪   redeemed:
        // uTokenAmount = amountIn
```

```
        // underlyingAmount = amountIn x exchangeRateCurrent
        uTokenAmount = amountIn;
        underlyingAmount = (amountIn * exchangeRate) / WAD;
    } else {
        // We get the current exchange rate and calculate the amount to be
↪   redeemed:
        // uTokenAmount = amountOut / exchangeRate
        // underlyingAmount = amountOut
        uTokenAmount = (amountOut * WAD) / exchangeRate;
        underlyingAmount = amountOut;
    }

    totalRedeemable -= underlyingAmount;
    _burn(msg.sender, uTokenAmount);

    IAssetManager assetManagerContract = IAssetManager(assetManager);
    if (!assetManagerContract.withdraw(underlying, msg.sender, underlyingAmount))
↪   revert WithdrawFailed();

    emit LogRedeem(msg.sender, amountIn, amountOut, underlyingAmount);
}
```

Same approach is in administrative removeReserves():

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/market/UToken.sol#L770-L784

```
function removeReserves(address receiver, uint256 reduceAmount)
    external
    override
    whenNotPaused
    nonReentrant
    onlyAdmin
{
    if (!accrueInterest()) revert AccrueInterestFailed();

    totalReserves -= reduceAmount;

    if (!IAssetManager(assetManager).withdraw(underlying, receiver,
↪   reduceAmount)) revert WithdrawFailed();

    emit LogReservesReduced(receiver, reduceAmount, totalReserves);
}
```

Tool used

Manual Review

Recommendation

Consider:

- either accounting for the actual amount withdrawn, i.e. return actual retrieved amount from withdraw(), move `IAssetManager(assetManager).withdraw` before other logic and operate this amount returned by withdraw() instead of `amount` initially requested in all the accounting logics above,
- or just require in withdraw() that actual amount withdrawn be equal to the requested one.

For the second option as fee on transfer tokens aren't in the main scope for credit lines functionality UNION covers, such a requirement will mean that if the `amount` is lacking the corresponding logic needs to be run again with a smaller one. But only amount actually withdrawn be accounted for, so the discrepancies be avoided:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L366-L369

```
        emit LogWithdraw(token, account, amount, remaining);


+       return remaining < (dustThreshold * amount) / WAD;
-       return true;
    }
```

Issue M-13: It's impossible to writing off any vouch fully for an outside actor

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/40

Found by

hyh, Ch_301, ctf_sec

Summary

UserManager's debtWriteOff() being called by a non-staker/borrower (after all overdue period thresholds pass) to remove bad debt will revert if the corresponding voucher is to be written off fully, i.e. has no trust left after the write-off.

In other words public debt writing off reverts in the full vouch removal case.

Vulnerability Detail

debtWriteOff() becomes public after all overdue delays pass, but when vouch write off is to be full, it calls vouch cancellation function, which aren't accommodated for it and reverts in this case. The reason is the more restrictive access controls in cancelVouch(), which do not utilize additional `availablebythepubliciftheloanisoverdue` logic of debtWriteOff().

As a workaround debtWriteOff() can be called with not full amount, leaving dust in the vouch, so almost all amount itself can be written off from the system. However, such vouches will end up open forever as long as the corresponding stakers/borrowers do not act.

Impact

Bad debt vouchers cannot be fully removed if staker/borrower isn't active, i.e. the immediate impact is unavailability of such public debt write-off.

Over time such bad debt vouchers with dust amounts will pile up, increasing operational costs for the functions that go through all existing vouches: UserManager's getCreditLimit() and getFrozenInfo(), UToken's borrow() and repayBorrow() via UserManager's updateLocked(). Over time this gas cost increase will apply to the substantial part of system users (i.e. as user link coverage grows and time passes the probability of a user affected by such zombie bad debt vouchers will slowly rise as well).

Long-term total impact can be up to the permanent funds freeze due to gas limit breaching and locking of the mentioned functions.

Code Snippet

debtWriteOff() is public when `block.number>lastRepay+overdueBlocks+maxOverdueBlocks`, but calls cancelVouch() when `vouch.trust==0`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L717-L778

```
/**
 *  @notice Write off a borrowers debt
 *  @dev    Used the stakers locked stake to write off the loan, transfering the
 *          Stake to the AssetManager and adjusting balances in the AssetManager
 *          and the UToken to repay the principal
 *  @dev    Emits {LogDebtWriteOff} event
 *  @param borrower address of borrower
 *  @param amount amount to writeoff
 */
function debtWriteOff(
    address staker,
    address borrower,
    uint96 amount
) external {
    if (amount == 0) revert AmountZero();
    uint256 overdueBlocks = uToken.overdueBlocks();
    uint256 lastRepay = uToken.getLastRepay(borrower);

    // This function is only callable by the public if the loan is overdue by
    // overdue blocks + maxOverdueBlocks. This stops the system being left with
    // debt that is overdue indefinitely and no ability to do anything about it.
    if (block.number <= lastRepay + overdueBlocks + maxOverdueBlocks) {
        if (staker != msg.sender) revert AuthFailed();
    }

    ...
    // update vouch trust amount
    vouch.trust -= amount;
    vouch.locked -= amount;
    ...

    if (vouch.trust == 0) {
        cancelVouch(staker, borrower);
    }
```

cancelVouch() allows `msg.sender` to be `borrower` or `staker` only:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L567-L578

```
/**
 *  @dev Remove voucher for memeber
 *  Can be called by either the borrower or the staker. It will remove the
 ↪  voucher from
 *  the voucher array by replacing it with the last item of the array and
 ↪  reseting the array
 *  size to -1 by poping off the last item
 *  Only callable by a member when the contract is not paused
 *  Emit {LogCancelVouch} event
 *  @param staker Staker address
 *  @param borrower borrower address
 */
function cancelVouch(address staker, address borrower) public
 ↪  onlyMember(msg.sender) whenNotPaused {
    if (staker != msg.sender && borrower != msg.sender) revert AuthFailed();
```

This will fail debtWriteOff() calls from any third parties, forcing them to leave dust in `vouch.trust`, choosing `amount=vouch.trust-dust` even when a vouch to be removed fully, which looks to be the frequent use case (bad debtors tend to fully utilize their credit lines).

Tool used

Manual Review

Recommendation

Consider the following, as an example:

Update cancelVouch() to allow for calls from self:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L577-L578

```
+   function cancelVouch(address staker, address borrower) public whenNotPaused {
+       bool selfCall = address(this) == msg.sender;
+       if ((!checkIsMember(msg.sender) && !selfCall) || (staker != msg.sender &&
 ↪  borrower != msg.sender && !selfCall)) revert AuthFailed();
-   function cancelVouch(address staker, address borrower) public
 ↪  onlyMember(msg.sender) whenNotPaused {
-       if (staker != msg.sender && borrower != msg.sender) revert AuthFailed();
```

Call it from itself in debtWriteOff() as a special case:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L776-L778

```
        if (vouch.trust == 0) {
+           IUserManager(address(this)).cancelVouch(staker, borrower);
-           cancelVouch(staker, borrower);
        }
```

Issue M-14: Asset manager's deposit, withdraw and re-balance function calls will get reverted when one of the adapters is broken or paused

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/37

Found by

ctf_sec, caventa

Summary

Asset manager's deposit, withdraw and rebalance function calls will get reverted when one of the adapters is broken or paused.

Vulnerability Detail

A given MoneyMarketAdapters can temporally or even permanently becomes mal-functioning (cannot deposit/withdraw) for all sorts of reasons. This results in all the other deposit, withdraw and rebalance calls to other adapters getting reverted.

Eg, Aave V2 Lending Pool can be paused, which will prevent multiple core functions that the Aave v2 depends on from working, including `deposit()` and `withdraw()`.

Impact

When Aave V2 Lending Pool is paused, deposit, withdraw, and rebalance function calls on other adapters will get reverted.

Code Snippet

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L290 https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L307 https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L529 https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L537 https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L357 https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L514

Tool used

Manual Review

Recommendation

Consider adding a try-catch clause to every code snippet written above so that when the pool is paused in AaveV2, deposit, withdrawal, and rebalance function calls are still allowed on other adapters.

For eg (For AssetManager.sol#L357) :

```
try moneyMarket.withdraw(token, account, withdrawAmount) {
 // Code added when there is no exception thrown
} catch {
// Code added when there is an exception thrown
}
```

Issue M-15: Maximal approvals remain for the AssetManager's adapters and tokens after removal

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/36

Found by

hyh, bin2chen, Jeiwan

Summary

While adding an adapter and a token to the AssetManager the system provides for the unlimited approvals for the current list of tokens and markets correspondingly, while the removal of the adapter and the token does not clear these approvals, which can be exploited thereafter as the contract balance does hold intermediary funds.

Vulnerability Detail

If the adapter or the token are being removed due to it being found to be malicious or contain a critical issue, treating downstream systems, currently there is no way to remove the exposure to it. This allowance can be exploited to drain AssetManager's balance.

That's a long term threat as market adapter or token being removed keep unlimited allowances forever as there is no mechanics to remove or limit those. I.e. some critical vulnerability can be found maybe substantially later in some adapter that was used by AssetManager and was removed long time ago. Its allowances remain and can be exploited to extract any funds held in the tokens from the list that was actual back then.

Impact

A malicious token or adapter can steal all the approved token's balances of the AssetManager. Some assets are routinely residing on the AssetManager's balance: as an example, when the funds, being deposited, can't be placed to the markets right away, they are left on the contract balance until next admin's rebalance() call.

An attacker can setup a bot that tracks this balance and exploit some vulnerability in an already removed adapter to fully drain the balance. Setting the severity to be **medium** due to the preconditions described.

Code Snippet

removeAdapter() leaves all the infinite approvals:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L436-L457

```
/**
 *  @dev Remove a adapter for the underlying lending protocol
 *  @param adapterAddress adapter address
 */
function removeAdapter(address adapterAddress) external override onlyAdmin {
    bool isExist = false;
    uint256 index;
    uint256 moneyMarketsLength = moneyMarkets.length;

    for (uint256 i = 0; i < moneyMarketsLength; i++) {
        if (adapterAddress == address(moneyMarkets[i])) {
            isExist = true;
            index = i;
            break;
        }
    }

    if (isExist) {
        moneyMarkets[index] = moneyMarkets[moneyMarketsLength - 1];
        moneyMarkets.pop();
    }
}
```

Similarly, removeToken() leaves all markets' unlimited approvals with the token intact:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L392-L414

```
/**
 *  @dev Remove a ERC20 token to support in AssetManager
 *  @param tokenAddress ERC20 token address
 */
function removeToken(address tokenAddress) external override onlyAdmin {
    bool isExist = false;
    uint256 index;
    uint256 supportedTokensLength = supportedTokensList.length;

    for (uint256 i = 0; i < supportedTokensLength; i++) {
        if (tokenAddress == address(supportedTokensList[i])) {
            isExist = true;
            index = i;
            break;
        }
    }
```

```
    if (isExist) {
        supportedTokensList[index] = supportedTokensList[supportedTokensLength -
↪   1];
        supportedTokensList.pop();
        supportedMarkets[tokenAddress] = false;
    }
}
```

AssetManager's balance aren't necessary empty as some funds are left until manual placement:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L261-L316

```
function deposit(address token, uint256 amount)
    external
    override
    whenNotPaused
    onlyAuth(token)
    nonReentrant
    returns (bool)
{
    IERC20Upgradeable poolToken = IERC20Upgradeable(token);
    if (amount == 0) revert AmountZero();

    if (!_isUToken(msg.sender, token)) {
        balances[msg.sender][token] += amount;
        totalPrincipal[token] += amount;
    }

    bool remaining = true;
    if (isMarketSupported(token)) {
        ...
    }

    if (remaining) {
        poolToken.safeTransferFrom(msg.sender, address(this), amount);
    }

    emit LogDeposit(token, msg.sender, amount);
```

Tool used

Manual Review

Recommendation

Consider introducing approvals clearing function and run it on adapter removal, for example:

```
+   /**
+    *  @dev Removal of the allowances for all underlying tokens
+    *  @param adapterAddress Address of adapter being removed
+    */
+   function removeApprovals(address adapterAddress) internal override {
+       uint256 supportedTokensLength = supportedTokensList.length;
+       for (uint256 i = 0; i < supportedTokensLength; ++i) {
+           IERC20Upgradeable poolToken =
↪  IERC20Upgradeable(supportedTokensList[i]);
+           poolToken.safeApprove(adapterAddress, 0);
+       }
+   }
```

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L436-L457

```
    /**
     *  @dev Remove a adapter for the underlying lending protocol
     *  @param adapterAddress adapter address
     */
    function removeAdapter(address adapterAddress) external override onlyAdmin {
        bool isExist = false;
        uint256 index;
        uint256 moneyMarketsLength = moneyMarkets.length;

        for (uint256 i = 0; i < moneyMarketsLength; i++) {
            if (adapterAddress == address(moneyMarkets[i])) {
                isExist = true;
                index = i;
                break;
            }
        }

        if (isExist) {
            moneyMarkets[index] = moneyMarkets[moneyMarketsLength - 1];
            moneyMarkets.pop();
+           removeApprovals(adapterAddress);
        }
    }
```

The same can be done for token removal, for example:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-cont

racts/contracts/asset/AssetManager.sol#L392-L414

```
    /**
     *  @dev Remove a ERC20 token to support in AssetManager
     *  @param tokenAddress ERC20 token address
     */
    function removeToken(address tokenAddress) external override onlyAdmin {
        bool isExist = false;
        uint256 index;
        uint256 supportedTokensLength = supportedTokensList.length;

        for (uint256 i = 0; i < supportedTokensLength; i++) {
            if (tokenAddress == address(supportedTokensList[i])) {
                isExist = true;
                index = i;
                break;
            }
        }

        if (isExist) {
            supportedTokensList[index] =
↪   supportedTokensList[supportedTokensLength - 1];
            supportedTokensList.pop();
            supportedMarkets[tokenAddress] = false;
+           removeTokenApprovals(tokenAddress);
        }
    }
```

There new removeTokenApprovals() function similarly to approveAllMarketsMax() cycles across all available markets, setting `IERC20Upgradeable(tokenAddress).safeApprove(address(moneyMarkets[i]),0)`.

Issue M-16: `AssetManager::withdraw` will not return false, when fail to send the withdraw amount

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/27

Found by

Ch_301, ctf_sec, cccz, yixxas, lemonmon

Summary

When `UserManager` or `UToken` calls on `AssetManager::withdraw`, the given `amount` should be transferred to `account`, either from `AssetManager` itself or by `moneyMarkets`. However, when there is not enough asset to be transferred from `AssetManager` itself or from `moneyMarkets`, it returns true, without reverting.

Since other contracts, who is using the `AssetManager::withdraw`, assumes that the `amount` is transferred, it causes problems such as:

- `UserManager::unstake`: the user might get less than the amount the user unstaked
- `UToken::borrow`: the user might get less than the amount the user borrowed
- `UToken::redeem`: the user might get less than the amount the user redeemed
- `UToken::removeReserves`: the admin might transfer out less than intended
- In some cases above will also result in accounting error

Vulnerability Detail

The `AssetManager::withdraw` function will transfer the given `amount` from `AssetManager` itself or from `moneyMarket`, since the values are distributed. The function keeps track of the remaining values to be transferred in `remaining` local variable. However, the function returns `true` even if there are some non zero `remaining` left to transfer.

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L328-L370

Other contracts who can call the function `AssetManager::withdraw` are `UserManager` and `UToken`. They assume that if the return from `AssetManager::withdraw` is `true`, the whole `amount` is transferred to the recipient, and update accounting accordingly. As the result, it will cause some cases that less `amount` is transferred out, yet the transaction does not fail.

The functions who uses `AssetManager::withdraw` are following:

- `UserManager::unstake`
  - the staker's `stakedAmount` will be decreased by `amount`, but the staker might get less than actually unstaked amount
- `UToken::borrow`: the user might get less than the amount the user borrowed
  - the user's `principal` will be increased by the borrowed `amount` and `fee`, but the user might get less than the asked `amount`
- `UToken::redeem`: the user might get less than the amount the user redeemed
  - the user's `UToken` is burned, but the user might get less than the burned amount.
- `UToken::removeReserves`: the admin might transfer out less than intended
  - the receiver might get less than what admin intended. The `totalReserves` might be reduced more than what was transferred out.

Impact

Users might get less amount transferred from the `AssetManager` than they should get

Code Snippet

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L328-L370

Tool used

Manual Review

Recommendation

Check the `remaining` to be transferred and return `false` if it is bigger than zero

Issue M-17: `Comptroller::withdrawRewards` accounting error results in incorrect inflation index

Source: https://github.com/sherlock-audit/2022-10-union-finance-judging/issues/26

Found by

Jeiwan, Lambda, dipp, lemonmon

Summary

In `Comptroller::withdrawRewards`, `totalFrozen` was subtracted twice from `totalStaked`, which will update `Comptroller::gInflationIndex` based on incorrect information. Also, if more than half of `totalStaked` is frozen, the `Comptroller::withdrawRewards` will revert, so no one can call `UserManager::stake` or `UserManager::unstake`.

Vulnerability Detail

In `Comptroller:withdrawRewards` calls `_getUserManagerState` and saves it as `userManagerState`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol?plain=1#L248

Note that returned value of `userManagerState.totalStaked` is equivalent to `userManager.totalStaked()-userManager.totalFrozen()`:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol?plain=1#L306-L317

However, in the `Comptroller::withdrawRewards` function, the returned value `userManagerState.totalStaked` will be subtracted by totalFrozen again:

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol?plain=1#L260-L261

So, the `totalStaked_` is equivalent to `userManager.totalStaked()-2*userManager.totalFrozen()`, which was used to calculate `gInflationIndex` in the line 261 of `Comptroller.sol`. It will result in incorrect update of the `gInflationIndex`.

Moreover, if more than half of total staked values are frozen, the line 260 in Comptroller.sol will revert from underflow. The `Comptroller::withdrawRewards` function is used in `UserManager::stake`, `UserManager::unstake` and `UserManager::withdrawRewards`, thus all of these function will stop working when the condition is met.

Impact

- Updated to incorrect `gInflationIndex`
- Revert when more than half of total staked is frozen

Code Snippet

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol?plain=1#L248 https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol?plain=1#L306-L317 https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol?plain=1#L260-L261

Tool used

Manual Review

Recommendation

the following line

https://github.com/sherlock-audit/2022-10-union-finance/blob/main/union-v2-contracts/contracts/token/Comptroller.sol?plain=1#L248

should be:

```
uint256 totalStaked_ = userManagerState.totalStaked;
```