SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Union
Prepared by: Sherlock
Lead Security Expert: hyh
Dates Audited: February 8 - February 11, 2023
Prepared on: February 27, 2023

Introduction

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

Scope

```
IAssetManager.sol
IDai.sol
IMarketRegistry.sol
IUDai.sol
IUnionToken.sol
Controller.sol
OpOwner.sol
UnionLens.sol
WadRayMath.sol
AaveV3Adapter.sol
AssetManager.sol
PureTokenAdapter.sol
IComptroller.sol
IInterestRateModel.sol
IMoneyMarketAdapter.sol
IUToken.sol
IUserManager.sol
FixedInterestRateModel.sol
MarketRegistry.sol
UDai.sol
UErc20.sol
UToken.sol
Comptroller.sol
OpConnector.sol
OpUNION.sol
Whitelistable.sol
UserManager.sol
UserManagerDAI.sol
UserManagerERC20.sol
UserManagerOp.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
6	3

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

hyh
seyni
weeeh_
ast3ros
ozooneth
chaduke
CodingNameKiki

Issue H-1: UserManager's cancelVouchInternal breaks up voucher accounting on voucher removal

Source: https://github.com/sherlock-audit/2023-02-union-judging/issues/41

Found by

hyh, seyni

Summary

_cancelVouchInternal() doesn't update `voucherIndices` array and incorrectly updates `vouchees` array (using `voucherIndexes` for `vouchees`, which are unrelated index-wise), messing up the vouchers accounting as a result.

Vulnerability Detail

Currently _cancelVouchInternal() gets voucher array index from `voucherIndexes`, name it `voucheeIdx` and apply to `vouchees`, which isn't correct as those are different arrays, their indices are independent, so such addressing messes up the accounting data.

Also, proper update isn't carried out for voucher indices themselves, although it is required for the future referencing which is used in all voucher related activities of the protocol.

Impact

Immediate impact is unavailability of vouch accounting logic 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

_cancelVouchInternal() correctly treats `vouchee` entry removal update, but fails to do the same for `vouchers` case (first in the code):

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L583-L620

```
function _cancelVouchInternal(address staker, address borrower) internal {
    Index memory removeVoucherIndex = voucherIndexes[borrower][staker];
    if (!removeVoucherIndex.isSet) revert VoucherNotFound();

    // Check that the locked amount for this vouch is 0
    Vouch memory vouch = vouchers[borrower][removeVoucherIndex.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
    {
        // Cache the last voucher
        Vouch memory lastVoucher = vouchers[borrower][vouchers[borrower].length
↪   - 1];
        // Move the lastVoucher to the index of the voucher we are removing
        vouchers[borrower][removeVoucherIndex.idx] = lastVoucher;
        // Pop the last vouch off the end of the vouchers array
        vouchers[borrower].pop();
        // Delete the voucher index for this borrower => staker pair
        delete voucherIndexes[borrower][staker];
        // Update the last vouchers coresponsing Vouchee item
        uint128 voucheeIdx = voucherIndexes[borrower][lastVoucher.staker].idx;
        vouchees[staker][voucheeIdx].voucherIndex =
↪   removeVoucherIndex.idx.toUint96();
    }

    // Update the vouchee entry for this borrower => staker pair
    {
        Index memory removeVoucheeIndex = voucheeIndexes[borrower][staker];
        // Cache the last vouchee
        Vouchee memory lastVouchee = vouchees[staker][vouchees[staker].length -
↪   1];
        // Move the last vouchee to the index of the removed vouchee
        vouchees[staker][removeVoucheeIndex.idx] = lastVouchee;
        // Pop the last vouchee off the end of the vouchees array
        vouchees[staker].pop();
        // Delete the vouchee index for this borrower => staker pair
        delete voucheeIndexes[borrower][staker];
        // Update the vouchee indexes to the new vouchee index
```

```
        voucheeIndexes[lastVouchee.borrower][staker].idx =
↪   removeVoucheeIndex.idx;
    }
```

Namely, `voucherIndexes` to be updated as `lastVoucher` has been moved and entries for its old position to be corrected on the spot. `vouchees` to be updated as well, but using correct `voucheeIndexes` index reference.

Tool used

Manual Review

Recommendation

Consider setting both `voucherIndexes` and `vouchees` items for the last staker:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L591-L605

```
        // 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
        {
            // Cache the last voucher
            Vouch memory lastVoucher =
↪   vouchers[borrower][vouchers[borrower].length - 1];
            // Move the lastVoucher to the index of the voucher we are removing
            vouchers[borrower][removeVoucherIndex.idx] = lastVoucher;
            // Pop the last vouch off the end of the vouchers array
            vouchers[borrower].pop();
            // Delete the voucher index for this borrower => staker pair
            delete voucherIndexes[borrower][staker];
-           // Update the last vouchers coresponsing Vouchee item
-           uint128 voucheeIdx =
↪   voucherIndexes[borrower][lastVoucher.staker].idx;
-           vouchees[staker][voucheeIdx].voucherIndex =
↪   removeVoucherIndex.idx.toUint96();
+           // Update the last vouchers coresponsing Vouchee item
+           vouchees[lastVoucher.staker][voucheeIndexes[borrower][lastVoucher.st
↪   aker]].voucherIndex = removeVoucherIndex.idx.toUint96();
+           // Update the voucher indexes of the moved pair to the new voucher
↪   index
+           voucherIndexes[borrower][lastVoucher.staker].idx =
↪   removeVoucherIndex.idx;
        }
```

Discussion

kingjacob

Is this not also a dupe of #4 ?

dmitriia

> Is this not also a dupe of #4 ?

#4 and #39 look to be partial reports of the #41/#31 issue, which basically has 2 parts.

#4 says '`vouchees[staker][voucheeIdx].voucherIndex = removeVoucherIndex.idx.toUint96()` doesn't look to be correct', not seeing the original `voucherIndexes` issue and not suggesting the proper fix for the `vouchees` one.

#39, on the other hand, only observes and suggests the fix for the `voucherIndexes` part, failing to notice the `vouchees` issue.

The most correct way to reflect this, as I see it, is to mark #4 and #39 as Mediums and non-dups as they are pointing to the not really related issues.

#41/#31 looks to be full dups and High as both investigate the situation fully (as I see it at the moment), providing fixes for the both parts.

hrishibhat

@kingjacob Agree with the @dmitriia . Considering #4 & #39 as separate mediums.

Issue H-2: Staker can manipulate lockedCoinAge and earn rewards in excess of the allowed maximum

Source: https://github.com/sherlock-audit/2023-02-union-judging/issues/26

Found by

hyh

Summary

Voucher can be counted arbitrary many times in staker's `lockedCoinAge`. If a voucher has maximized its trust then its `locked` is added to the `lockedCoinAge` each time fully as its `lastUpdated` is kept intact. This provides a surface to grow `lockedCoinAge` as big as an attacker wants, increasing it by `current_block_difference * vouch.locked` on each transaction.

Vulnerability Detail

`vouch.lastUpdated` is used for `stakers[vouch.staker].lockedCoinAge` accumulator and should be updated whenever its current state is recorded there. However, now it is not always the case as `lockedCoinAge` is updated in updateLocked() for every vouch until `remaining` is hit, while for the vouches, whose trust is maxxed in the borrow case (`lock == true`), such `lastUpdated` renewal doesn't take place.

Impact

An attacker will be able to steal all UNION rewards at once as with `lockedCoinAge` arbitrary big the reward multiplier, which isn't controlled to the upside, will be arbitrary big as well and an attacker will claim all available UNION in the contract.

Code Snippet

updateLocked() accumulates `stakers[vouch.staker].lockedCoinAge` before vouch logic is run.

However, `lastUpdated` isn't always updated, which leads to ability to grow the accumulator when `lock == true` and `vouch.trust == vouch.locked`:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L838-L870

```
uint256 lastWithdrawRewards = getLastWithdrawRewards[vouch.staker];
stakers[vouch.staker].lockedCoinAge +=
    (block.number - _max(lastWithdrawRewards, uint256(vouch.lastUpdated))) *
    uint256(vouch.locked);
```

```
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( ..., true)` is called on each borrow() with `amount > minBorrow`:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/market/UToken.sol#L525-L568

```
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();

    ...

    // 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, (actualAmount +
↪   fee).toUint96(), true);
```

So, whenever it's `lock == true`, `vouch.trust == vouch.locked`, the `stakers[vouch.staker].lockedCoinAge += (block.number - _max(lastWithdrawRewards, uint256(vouch.lastUpdated))) * uint256(vouch.locked)` will be counted for such a voucher with old `vouch.lastUpdated` as many times as updateLocked() run, which can be orchestrated easily enough.

Suppose Bob the staker has a vouch with trust maxxed, i.e. `vouch.trust = vouch.locked = 10k DAI`. He can setup a second borrower being his own account, some minimal trust, then can run min borrow many, many times, gaining huge `stakers[vouch.staker].lockedCoinAge` as `vouch.lastUpdated` aren't updated and `lockedCoinAge` grows with a positive `some_number_of_blocks * 10k DAI` number each time Bob borrows `1 DAI` via his second borrower.

`some_number_of_blocks` here is `(block.number - _max(lastWithdrawRewards, uint256(vouch.lastUpdated)))` and it can be any as attack is viable as long as it is positive. Bob can wait for some time before proceeding to grow this number of blocks to make the attack bit more effective.

With `effectiveLocked` arbitrary big Bob will be able to obtain arbitrary big `lendingRatio` and _getRewardsMultiplier() result:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L433-L445

```
function _getRewardsMultiplier(UserManagerAccountState memory user) internal
↪   pure returns (uint256) {
    if (user.isMember) {
        if (user.effectiveStaked == 0) {
            return memberRatio;
        }

        uint256 lendingRatio = user.effectiveLocked.wadDiv(user.effectiveStaked);

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

This will issue an arbitrary big amount of rewards to him, so Bob can effectively steal all UNION from the reward contract:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L341-L364

```
function _calculateRewardsByBlocks(
    address account,
    address token,
    uint256 pastBlocks,
    Info memory userInfo,
    uint256 totalStaked,
    UserManagerAccountState memory user
) internal view returns (uint256) {
    uint256 startInflationIndex = users[account][token].inflationIndex;

    if (user.effectiveStaked == 0 || totalStaked == 0 || startInflationIndex ==
↪   0 || pastBlocks == 0) {
        return 0;
    }

    uint256 rewardMultiplier = _getRewardsMultiplier(user);

    uint256 curInflationIndex = _getInflationIndexNew(totalStaked, pastBlocks);

    if (curInflationIndex < startInflationIndex) revert InflationIndexTooSmall();

    return
        userInfo.accrued +
        (curInflationIndex -
↪   startInflationIndex).wadMul(user.effectiveStaked).wadMul(rewardMultiplier);
}
```

Tool used

Manual Review

Recommendation

Consider updating `vouch.lastUpdated` every time it participated in `lockedCoinAge` accumulator:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L838-L870

```
            uint256 lastWithdrawRewards = getLastWithdrawRewards[vouch.staker];
            stakers[vouch.staker].lockedCoinAge +=
                (block.number - _max(lastWithdrawRewards,
↪   uint256(vouch.lastUpdated))) *
                uint256(vouch.locked);
+           vouch.lastUpdated = uint64(block.number);
            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);
            }
```

It is used only in _getEffectiveAmounts() as an addition to `coinAge.lockedCoinAge` already recorded:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L927-L929

```
    uint256 lastUpdateBlock = _max(coinAge.lastWithdrawRewards,
↪   uint256(vouch.lastUpdated));
    coinAge.lockedCoinAge += (block.number - lastUpdateBlock) * locked;
}
```

Issue H-3: Staker can perform rewards withdrawal every overdueBlocks less 1 and have zero frozenCoinAge

Source: https://github.com/sherlock-audit/2023-02-union-judging/issues/25

Found by

hyh

Summary

Time difference accounted for in _getEffectiveAmounts() that is used for rewards computation is based on the lesser amount of time passed since last reward withdrawal and borrower's last repay. In order to maximize the rewards a staker can withdraw rewards every `uToken.overdueBlocks() - 1`, so that difference is always be less than overdue threshold and none of staker's loans be deemed overdue no matter how long ago their last payments were.

Vulnerability Detail

Effective amounts are counted to determine the reward multiplier and amount, and the frozen part of these amounts corresponds to the bad debt. However, the time period check for the inclusion of a borrower to the `frozenCoinAge` is performed as `overdueBlocks < repayDiff`, where `repayDiff = block.number - _max(lastRepay, coinAge.lastWithdrawRewards)`, and `lastWithdrawRewards = getLastWithdrawRewards[stakerAddress]` is staker controlled.

So this `repayDiff` can be always kept below `overdueBlocks` via calling rewards gathering frequent enough, once per `overdueBlocks - 1` or sooner.

Also notice that _getCoinAge() returned `coinAge.lockedCoinAge` is updated on overdue debt repayment only and is zero when no such repayments were made.

As an example, a staker can lend to themselves, pay no interest, call withdrawRewards() once per `overdueBlocks - 1`, gathering full rewards without any penalty. Or just observing bad debt start to do it in order to remove its impact on rewards.

In any case this provides a way to remove `frozenCoinAge` based bad debt penalty altogether.

Impact

Excess UNION rewards emissions 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

_getEffectiveAmounts() accumulates `frozenCoinAge` only when `uToken.overdueBlocks() = overdueBlocks < repayDiff = block.number - _max(lastRepay, coinAge.lastWithdrawRewards)`, treating it as zero for that voucher otherwise:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L892-L938

```
function _getEffectiveAmounts(address stakerAddress, uint256 pastBlocks)
...
{
    uint256 memberTotalFrozen = 0;
    CoinAge memory coinAge = _getCoinAge(stakerAddress);

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

        uint256 lastRepay = uToken.getLastRepay(vouchee.borrower);
        uint256 repayDiff = block.number - _max(lastRepay,
↪   coinAge.lastWithdrawRewards);
        uint256 locked = uint256(vouch.locked);

        if (overdueBlocks < repayDiff && (coinAge.lastWithdrawRewards != 0 ||
↪   lastRepay != 0)) {
            memberTotalFrozen += locked;
            if (pastBlocks >= repayDiff) {
                coinAge.frozenCoinAge += (locked * repayDiff);
            } else {
                coinAge.frozenCoinAge += (locked * pastBlocks);
            }
        }
```

```
        uint256 lastUpdateBlock = _max(coinAge.lastWithdrawRewards,
↪   uint256(vouch.lastUpdated));
        coinAge.lockedCoinAge += (block.number - lastUpdateBlock) * locked;
    }

    return (
        // staker's total effective staked = (staked coinage - frozen coinage) /
↪   (# of blocks since last reward claiming)
        coinAge.diff == 0 ? 0 : (coinAge.stakedCoinAge - coinAge.frozenCoinAge)
↪   / coinAge.diff,
        // effective locked amount = (locked coinage - frozen coinage) / (# of
↪   blocks since last reward claiming)
        coinAge.diff == 0 ? 0 : (coinAge.lockedCoinAge - coinAge.frozenCoinAge)
↪   / coinAge.diff,
        memberTotalFrozen
    );
}
```

_getCoinAge() returns `getLastWithdrawRewards[stakerAddress]` as `lastWithdrawRewards`, i.e. that's staker controlled, and `block.number - _max(lastWithdrawRewards, uint256(staker.lastUpdated))` as `diff`, `frozenCoinAge[stakerAddress]` as `frozenCoinAge`, which serves as a base for frozen accumulator:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L1072-L1087

```
function _getCoinAge(address stakerAddress) private view returns (CoinAge
↪   memory) {
    Staker memory staker = stakers[stakerAddress];

    uint256 lastWithdrawRewards = getLastWithdrawRewards[stakerAddress];
    uint256 diff = block.number - _max(lastWithdrawRewards,
↪   uint256(staker.lastUpdated));

    CoinAge memory coinAge = CoinAge({
        lastWithdrawRewards: lastWithdrawRewards,
        diff: diff,
        stakedCoinAge: staker.stakedCoinAge + diff *
↪   uint256(staker.stakedAmount),
        lockedCoinAge: staker.lockedCoinAge,
        frozenCoinAge: frozenCoinAge[stakerAddress]
    });

    return coinAge;
}
```

While across the protocol `frozenCoinAge[stakerAddress]` is accumulated only on onRepayBorrow():

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L995-L1015

```
/**
 * @dev Update the frozen info by the utoken repay
 * @param borrower Borrower address
 */
function onRepayBorrow(address borrower) external {
    if (address(uToken) != msg.sender) revert AuthFailed();

    uint256 overdueBlocks = uToken.overdueBlocks();

    uint256 vouchersLength = vouchers[borrower].length;
    uint256 lastRepay = 0;
    uint256 diff = 0;
    for (uint256 i = 0; i < vouchersLength; i++) {
        Vouch memory vouch = vouchers[borrower][i];
        lastRepay = uToken.getLastRepay(borrower);
        diff = block.number - lastRepay;
        if (overdueBlocks < diff) {
            frozenCoinAge[vouch.staker] += uint256(vouch.locked) * diff;
        }
    }
}
```

onRepayBorrow() is called only by _repayBorrowFresh() and only when the debt is overdue:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/market/UToken.sol#L601-L638

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

    if (repayAmount >= interest) {
        ...

        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).onRepayBorrow(borrower);
```

```
        }
```

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/market/UToken.sol#L389-L395

```
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;
    }
}
```

I.e. if a staker's borrowers didn't repaid at all, or repaid without overdue for some period and then became bad debt, making no repayments after some date, then `frozenCoinAge[stakerAddress]` is zero as no overdue repayments were ever made.

Furthermore, staker can call withdrawRewards() to update `getLastWithdrawRewards[stakerAddress]` and keep `diff = block.number - _max(lastWithdrawRewards, uint256(staker.lastUpdated))` below current `uToken.overdueBlocks()`, keeping their frozen counter zero.

Tool used

Manual Review

Recommendation

As an example approach, consider introducing lifetime effective staked, locked and frozen accumulators, i.e. do not resetting them on each rewards claim, and using the difference vs the last claim point instead.

This way total staked, locked, frozen counters will be kept since staker inception, but only the last yet unrewarded period will be used to calculate each next portion of the rewards.

Manipulation surface described will not be viable as withdrawRewards() will only determine the reward period and will not be resetting the whole reward accounting.

Issue M-1: # [M-04] voucherIndexes is not updated when member cancel the voucher

Source: https://github.com/sherlock-audit/2023-02-union-judging/issues/39

Found by

ast3ros

Summary

When staker or borrower cancel the voucher, the `voucherIndexes` for the lastVoucher is not updated.

Vulnerability Details

The voucherIndexes is not updated when a voucher is cancelled.

Impact

It leads to incorrect index for the last voucher, which points to another voucher with different staker. It could be a point to be exploit by malicious members.

Code Snippet

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L593-L605

Tool used

Manual

Recommendation

Add update voucherIndexes in the `_cancelVouchInternal` function, below line 604 of UToken.sol:

```
voucherIndexes[borrower][lastVoucher.staker] = removeVoucherIndex.Idx
```

Issue M-2: Users can lose their staking rewards.

Source: https://github.com/sherlock-audit/2023-02-union-judging/issues/29

Found by

CodingNameKiki

Summary

By following the steps described in `Vulnerability Detail`, user is able to lose all of his staking rewards.

Vulnerability Detail

The issue occurs in the following steps described below:

1. Kiki calls the function `unstake` and unstakes all of his funds, as a result the internal function `_updateStakedCoinAge` is called to update his staked coin age till the current block.

```
contracts/user/UserManager.sol

711:  function unstake(uint96 amount) external whenNotPaused nonReentrant {
712:        Staker storage staker = stakers[msg.sender];
713:
714:        // Stakers can only unstaked stake balance that is unlocked. Stake
↪   balance
715:        // becomes locked when it is used to underwrite a borrow.
716:        if (staker.stakedAmount - staker.locked < amount) revert
↪   InsufficientBalance();
717:
718:        comptroller.withdrawRewards(msg.sender, stakingToken);
719:
720:        uint256 remaining =
↪   IAssetManager(assetManager).withdraw(stakingToken, msg.sender, amount);
721:        if (uint96(remaining) > amount) {
722:            revert AssetManagerWithdrawFailed();
723:        }
724:        uint96 actualAmount = amount - uint96(remaining);
725:
726:        _updateStakedCoinAge(msg.sender, staker);
727:        staker.stakedAmount -= actualAmount;
728:        totalStaked -= actualAmount;
729:
730:        emit LogUnstake(msg.sender, actualAmount);
```

```
731:     }
```

```
contracts/user/UserManager.sol

1064:   function _updateStakedCoinAge(address stakerAddress, Staker storage
↪  staker) private {
1065:         uint64 currentBlock = uint64(block.number);
1066:         uint256 lastWithdrawRewards = getLastWithdrawRewards[stakerAddress];
1067:         uint256 blocksPast = (uint256(currentBlock) -
↪  _max(lastWithdrawRewards, uint256(staker.lastUpdated)));
1068:         staker.stakedCoinAge += blocksPast * uint256(staker.stakedAmount);
1069:         staker.lastUpdated = currentBlock;
1070:     }
```

2. After that Kiki calls the function `withdrawRewards` in order to withdraw his staking rewards. Everything executes fine, but the contract lacks union tokens and can't transfer the tokens to Kiki, so the else statement is triggered and the amount of tokens is added to his accrued balance, so he can still be able to withdraw them after.

```
contracts/token/Comptroller.sol

224:   function withdrawRewards(address account, address token) external override
↪  whenNotPaused returns (uint256) {
225:         IUserManager userManager = _getUserManager(token);
226:
227:         // Lookup account state from UserManager
228:         (UserManagerAccountState memory user, Info memory userInfo, uint256
↪  pastBlocks) = _getUserInfo(
229:             userManager,
230:             account,
231:             token,
232:             0
233:         );
234:
235:         // Lookup global state from UserManager
236:         uint256 globalTotalStaked = userManager.globalTotalStaked();
237:
238:         uint256 amount = _calculateRewardsByBlocks(account, token,
↪  pastBlocks, userInfo, globalTotalStaked, user);
239:
240:         // update the global states
241:         gInflationIndex = _getInflationIndexNew(globalTotalStaked,
↪  block.number - gLastUpdatedBlock);
242:         gLastUpdatedBlock = block.number;
243:         users[account][token].updatedBlock = block.number;
244:         users[account][token].inflationIndex = gInflationIndex;
```

```
245:         if (unionToken.balanceOf(address(this)) >= amount && amount > 0) {
246:             unionToken.safeTransfer(account, amount);
247:             users[account][token].accrued = 0;
248:             emit LogWithdrawRewards(account, amount);
249:
250:             return amount;
251:         } else {
252:             users[account][token].accrued = amount;
253:             emit LogWithdrawRewards(account, 0);
254:
255:             return 0;
256:         }
257:     }
```

3. This is where the issue occurs, next time Kiki calls the function `withdrawRewards`, he is going to lose all of his rewards.

Explanation of how this happens:

- First the internal function _getUserInfo will return the struct `UserManagerAccountState memory user`, which contains zero amount for effectiveStaked, because Kiki unstaked all of his funds and already called the function withdrawRewards once. This happens because Kiki has `stakedAmount = 0, stakedCoinAge = 0, lockedCoinAge = 0, frozenCoinAge = 0`.

```
(UserManagerAccountState memory user, Info memory userInfo, uint256 pastBlocks)
↪  = _getUserInfo(
            userManager,
            account,
            token,
            0
        );
```

- The cache `uint256 amount` will have a zero value because of the if statement applied in the internal function `_calculateRewardsByBlocks`, the if statement will be triggered as Kiki's effectiveStaked == 0, and as a result the function will return zero.

```
uint256 amount = _calculateRewardsByBlocks(account, token, pastBlocks, userInfo,
↪  globalTotalStaked, user);
```

```
if (user.effectiveStaked == 0 || totalStaked == 0 || startInflationIndex == 0 ||
↪  pastBlocks == 0) {
            return 0;
        }
```

- Since the cache `uint256 amount` have a zero value, the if statement in the

function `withdrawRewards` will actually be ignored because of `&& amount > 0`. And the else statement will be triggered, which will override Kiki's accrued balance with "amount", which is actually zero. As a result Kiki will lose his rewards.

```
if (unionToken.balanceOf(address(this)) >= amount && amount > 0) {
            unionToken.safeTransfer(account, amount);
            users[account][token].accrued = 0;
            emit LogWithdrawRewards(account, amount);

            return amount;
        } else {
            users[account][token].accrued = amount;
            emit LogWithdrawRewards(account, 0);

            return 0;
        }
```

Impact

The impact here is that users can lose their staking rewards.

To understand the scenario which is described in `Vulnerability Detail`, you'll need to know how the codebase works. Here in the impact section, I will describe in little more details and trace the functions.

The issue occurs in 3 steps like described in `Vulnerability Detail`:

1. User unstakes all of his funds.
2. Then he calls the function `withdrawRewards` in order to withdraw his rewards, everything executes fine but the contract lacks union tokens, so instead of transferring the tokens to the user, they are added to his accrued balance so he can still withdraw them after.
3. The next time the user calls the function `withdrawRewards` in order to withdraw his accrued balance of tokens, he will lose all of his rewards.

Explanation in details:

1. User unstakes all of his funds by calling the function `unstake`.

- His stakedAmount will be reduced to zero in the struct `Staker`.
- His stakedCoinAge will be updated to the current block with the internal function `_updateStakedCoinAge`.

2. Then he calls the function withdrawRewards in order to withdraw his rewards, everything executes fine but the contract lacks union tokens, so instead of

transferring the tokens to the user, they are added to his accrued balance so he can still withdraw them after.

- User's stakedCoinAge, lockedCoinAge and frozenCoinAge are reduced to zero in the function `onWithdrawRewards`.

3. The next time the user calls the function `withdrawRewards` in order to withdraw his accrued balance of tokens, he will lose all of his rewards.

- In order to withdraw his accrued rewards stored in his struct balance `Info`. He calls the function `withdrawRewards` again and this is where the issue occurs, as the user has `stakedAmount = 0, stakedCoinAge = 0, lockedCoinAge = 0, frozenCoinAge = 0` .
- Duo to that the outcome of the function _getCoinAge, which returns a memory struct of CoinAge to the function `_getEffectiveAmounts` will look like this:
- As a result the function `_getEffectiveAmounts` will return zero values for effectiveStaked and effectiveLocked to the function `onWithdrawRewards`.
- After that the function `withdrawRewards` caches the returning value from the internal function `_calculateRewardsByBlocks`. What happens is that in the function `_calculateRewardsByBlocks` the if statement is triggered because the user's effectiveStaked == 0. As a result the internal function will return 0 and the cache `uint256 amount` will equal zero.
- Since the cache `uint256 amount` have a zero value, the if statement in the function `withdrawRewards` will actually be ignored because of `&& amount > 0`. And the else statement will be triggered, which will override Kiki's accrued balance with "amount", which is actually zero.

Below you can see the functions which are invoked starting from the function `_getUserInfo`:

```
(UserManagerAccountState memory user, Info memory userInfo, uint256 pastBlocks)
↪  = _getUserInfo(
            userManager,
            account,
            token,
            0
        );
```

```
function _getUserInfo(
        IUserManager userManager,
        address account,
        address token,
        uint256 futureBlocks
    ) internal returns (UserManagerAccountState memory user, Info memory
↪  userInfo, uint256 pastBlocks) {
```

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

        pastBlocks = block.number - lastUpdatedBlock + futureBlocks;

        (user.effectiveStaked, user.effectiveLocked, user.isMember) =
↪   userManager.onWithdrawRewards(
            account,
            pastBlocks
        );
    }
```

```
function onWithdrawRewards(address staker, uint256 pastBlocks)
        external
        returns (
            uint256 effectiveStaked,
            uint256 effectiveLocked,
            bool isMember
        )
    {
        if (address(comptroller) != msg.sender) revert AuthFailed();
        uint256 memberTotalFrozen = 0;
        (effectiveStaked, effectiveLocked, memberTotalFrozen) =
↪   _getEffectiveAmounts(staker, pastBlocks);
        stakers[staker].stakedCoinAge = 0;
        stakers[staker].lastUpdated = uint64(block.number);
        stakers[staker].lockedCoinAge = 0;
        frozenCoinAge[staker] = 0;
        getLastWithdrawRewards[staker] = block.number;

        uint256 memberFrozenBefore = memberFrozen[staker];
        if (memberFrozenBefore != memberTotalFrozen) {
            memberFrozen[staker] = memberTotalFrozen;
            totalFrozen = totalFrozen - memberFrozenBefore + memberTotalFrozen;
        }

        isMember = stakers[staker].isMember;
    }
```

```
function _getEffectiveAmounts(address stakerAddress, uint256 pastBlocks)
        private
        view
        returns (
            uint256,
```

```
            uint256,
            uint256
        )
    {
        uint256 memberTotalFrozen = 0;
        CoinAge memory coinAge = _getCoinAge(stakerAddress);

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

            uint256 lastRepay = uToken.getLastRepay(vouchee.borrower);
            uint256 repayDiff = block.number - _max(lastRepay,
↪   coinAge.lastWithdrawRewards);
            uint256 locked = uint256(vouch.locked);

            if (overdueBlocks < repayDiff && (coinAge.lastWithdrawRewards != 0
↪   || lastRepay != 0)) {
                memberTotalFrozen += locked;
                if (pastBlocks >= repayDiff) {
                    coinAge.frozenCoinAge += (locked * repayDiff);
                } else {
                    coinAge.frozenCoinAge += (locked * pastBlocks);
                }
            }

            uint256 lastUpdateBlock = _max(coinAge.lastWithdrawRewards,
↪   uint256(vouch.lastUpdated));
            coinAge.lockedCoinAge += (block.number - lastUpdateBlock) * locked;
        }

        return (
            // staker's total effective staked = (staked coinage - frozen
↪   coinage) / (# of blocks since last reward claiming)
            coinAge.diff == 0 ? 0 : (coinAge.stakedCoinAge -
↪   coinAge.frozenCoinAge) / coinAge.diff,
            // effective locked amount = (locked coinage - frozen coinage) / (#
↪   of blocks since last reward claiming)
            coinAge.diff == 0 ? 0 : (coinAge.lockedCoinAge -
↪   coinAge.frozenCoinAge) / coinAge.diff,
            memberTotalFrozen
```


```
        );
    }
```

```
function _getCoinAge(address stakerAddress) private view returns (CoinAge
↪  memory) {
        Staker memory staker = stakers[stakerAddress];

        uint256 lastWithdrawRewards = getLastWithdrawRewards[stakerAddress];
        uint256 diff = block.number - _max(lastWithdrawRewards,
↪  uint256(staker.lastUpdated));

        CoinAge memory coinAge = CoinAge({
            lastWithdrawRewards: lastWithdrawRewards,
            diff: diff,
            stakedCoinAge: staker.stakedCoinAge + diff *
↪  uint256(staker.stakedAmount),
            lockedCoinAge: staker.lockedCoinAge,
            frozenCoinAge: frozenCoinAge[stakerAddress]
        });

        return coinAge;
    }
```

Below you can see the function _calculateRewardsByBlocks:

```
function _calculateRewardsByBlocks(
        address account,
        address token,
        uint256 pastBlocks,
        Info memory userInfo,
        uint256 totalStaked,
        UserManagerAccountState memory user
    ) internal view returns (uint256) {
        uint256 startInflationIndex = users[account][token].inflationIndex;

        if (user.effectiveStaked == 0 || totalStaked == 0 || startInflationIndex
↪  == 0 || pastBlocks == 0) {
            return 0;
        }

        uint256 rewardMultiplier = _getRewardsMultiplier(user);

        uint256 curInflationIndex = _getInflationIndexNew(totalStaked,
↪  pastBlocks);

        if (curInflationIndex < startInflationIndex) revert
↪  InflationIndexTooSmall();
```

```
        return
            userInfo.accrued +
            (curInflationIndex -
↪   startInflationIndex).wadMul(user.effectiveStaked).wadMul(rewardMultiplier);
    }
```

Code Snippet

Function `unstake` - https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L711-L731

Function `_updateStakedCoinAge` - https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L1064-L1070

Function `withdrawRewards` - https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L224-L257

Function `_getUserInfo` - https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L311-L329

Function `onWithdrawRewards` - https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L969-L993

Function `_getEffectiveAmounts` - https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L892-L938

Function `_getCoinAge` - https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L1072-L1087

Function `_calculateRewardsByBlocks` - https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L341-L364

Tool used

Manual Review

Recommendation

One way of fixing this problem, that I can think of is to refactor the function _calculateRewardsByBlocks. First the function _calculateRewardsByBlocks will revert if `(totalStaked == 0 || startInflationIndex == 0 || pastBlocks == 0)`. Second new if statement is created, which is triggered if `user.effectiveStaked == 0`.

- if `userInfo.accrued == 0`, it will return 0.
- if `userInfo.accrued != 0`, it will return the accrued balance.

```
function _calculateRewardsByBlocks(
        address account,
        address token,
        uint256 pastBlocks,
        Info memory userInfo,
        uint256 totalStaked,
        UserManagerAccountState memory user
    ) internal view returns (uint256) {
        uint256 startInflationIndex = users[account][token].inflationIndex;

        if (totalStaked == 0 || startInflationIndex == 0 || pastBlocks == 0) {
            revert ZeroNotAllowed();
        }

        if (user.effectiveStaked == 0) {
           if (userInfo.accrued == 0) return 0;
           else return userInfo.accrued
        }

        uint256 rewardMultiplier = _getRewardsMultiplier(user);

        uint256 curInflationIndex = _getInflationIndexNew(totalStaked,
↪   pastBlocks);

        if (curInflationIndex < startInflationIndex) revert
↪   InflationIndexTooSmall();

        return
            userInfo.accrued +
            (curInflationIndex -
↪   startInflationIndex).wadMul(user.effectiveStaked).wadMul(rewardMultiplier);
    }
```

Discussion

kingjacob

This is valid but medium severity as its inaccurate rewards not a loss of funds.

hrishibhat

Considering this issue a valid medium as there is a precondition of an underfunded contract for this to occur which is highly unlikely.

Issue M-3: Market adapter removal corrupts withdraw sequence

Source: https://github.com/sherlock-audit/2023-02-union-judging/issues/24

Found by

hyh

Summary

Withdraw sequence entry is being deleted in AssetManager's removeAdapter() using incorrect index, so the sequence array becomes corrupted and withdrawals can end up unavailable.

Vulnerability Detail

`index` used for adapter removal is determined for `moneyMarkets` array, but applied to `withdrawSeq` as well, while in there indices do differ.

Impact

As withdrawals logic are based on withdraw sequence array, the withdrawals can end up unavailable. For example, if 95% of the funds are held in a market, whose entry corresponded to the `index` removed, withdrawals will be frozen for an arbitrary time for the whole protocol until `withdrawSeq` be manually restored.

Code Snippet

removeAdapter() determines the `index` for `moneyMarkets` array:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L435-L463

```
/**
 *  @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 = 0;
    uint256 moneyMarketsLength = moneyMarkets.length;
    for (uint256 i = 0; i < moneyMarketsLength; i++) {
        if (adapterAddress == address(moneyMarkets[i])) {
            isExist = true;
            index = i;
```

```
            break;
        }
    }

    if (isExist) {
        for (uint256 i = 0; i < supportedTokensList.length; i++) {
            if (moneyMarkets[index].getSupply(supportedTokensList[i]) >= 10000)
↪   revert RemainingFunds(); //ignore the dust
        }
        moneyMarkets[index] = moneyMarkets[moneyMarketsLength - 1];
        moneyMarkets.pop();

        withdrawSeq[index] = withdrawSeq[withdrawSeq.length - 1];
        withdrawSeq.pop();

        _removeTokenApprovals(adapterAddress);
    }
}
```

But `withdrawSeq` is a permutation of `moneyMarkets`, so their indices are independent and do not correspond to each other:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L136-L143

```
/**
 *  @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 != moneyMarkets.length) revert NotParity();
    withdrawSeq = newSeq;
}
```

So `index` in `withdrawSeq` being deleted do not generally correspond to the adapter, i.e. some other market end up being removed:

https://github.com/unioncredit/union-v2-contracts/blob/49d1a7261a7be20fe77b91a8a73e3cba8bc5bda5/contracts/asset/AssetManager.sol#L464-L465

```
withdrawSeq[index] = withdrawSeq[withdrawSeq.length - 1];
withdrawSeq.pop();
```

Tool used

Manual Review

Recommendation

Consider repeating the entry finding logic for `withdrawSeq`, for example:

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/asset/AssetManager.sol#L435-L463

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

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

            for (uint256 i = 0; i < supportedTokensList.length; i++) {
                if (moneyMarkets[index].getSupply(supportedTokensList[i]) >=
 ↪  10000) revert RemainingFunds(); //ignore the dust
            }
            moneyMarkets[index] = moneyMarkets[moneyMarketsLength - 1];
            moneyMarkets.pop();

-           withdrawSeq[index] = withdrawSeq[withdrawSeq.length - 1];
+           withdrawSeq[indexSeq] = withdrawSeq[withdrawSeq.length - 1];
            withdrawSeq.pop();

            _removeTokenApprovals(adapterAddress);
```

```
        }
    }
```

Issue M-4: Attackers can call UToken.redeem() and drain the funds in assetManager

Source: https://github.com/sherlock-audit/2023-02-union-judging/issues/12

Found by

chaduke

Summary

Attackers can call `UToken.redeem()` and drain the funds in assetManager, taking advantage of the following vulnerability: due to round error, it is possible that `uTokenAmount = 0`; the `redeem()`function does not check whether `uTokenAmount = 0` and will redeem the amount of `underlyingAmount` even when zero uTokens are burned.

Vulnerability Detail

Consider the following attack scenario:

1) Suppose `exchangeRate = 1000 WAD`, that is each utoken exchanges for 1000 underlying tokens.
2) Attacker B calls `redeem(0, 999)`, then the else-part of the following code will get executed:

```
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;
        }
```

3) we have `uTokenAmount = 999*WAD/1000WAD = 0`, and `underlyingAmount = 999`.

4) Since `redeeem()` does not check whether `uTokenAmount = 0` and the function will proceed. When finished, the attacker will get 999 underlying tokens, but burned no utokens. He stole 999 underlying tokens.
5) The attacker can accomplish draining the `assetManger` by writing a malicious contract/function using a loop to run `redeem(0, exchangeRate/WAD-1)` multiple times (as long as not running of gas) and will be able to steal more funds in one SINGLE transaction. Running this transaction a few times will drain `assetManager` easily. This attack will be successful when `exchangeRate/WAD-1 > 0`. Here we need to consider that `exchangeRate` might change due to the decreasing of `totalReeemable`. So in each iteration, when we call `redeem(0, exchangeRate/WAD-1)`, the second argument is recalculated.

Impact

An attacker can keep calling redeem() and drain the funds in `assetManager`.

Code Snippet

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/market/UToken.sol#L737-L771

Tool used

Remix

Manual Review

Recommendation

Revise `redeem()` so that it will revert when `uTokenAmount = 0`.

Discussion

kingjacob

points valid, but not easy to increase exchangeRate exponentially. becasue exchangeRate starts from 1e18, 10x of exchangeRate means UToken value, by accruing interests, has increased 10x comparing to the underlying DAI. and the attack can profit 9 DAI in that case)

hrishibhat

Agree with the Sponsor on the severity of the issue as exchangeRate cannot be increase exponentially. Considering this a valid medium.

Issue M-5: The `exchangeRateStored()` function allows front-running on repayments

Source: https://github.com/sherlock-audit/2023-02-union-judging/issues/8

Found by

ozoooneth

Summary

The `exchangeRateStored()` function allows to perform front-running attacks when a repayment is being executed.

Vulnerability Detail

Since `_repayBorrowFresh()` increases `totalRedeemable` value which affects in the final exchange rate calculation used in functions such as `mint()` and `redeem()`, an attacker could perform a front-run to any repayment by minting `UTokens` beforehand, and redeem these tokens after the front-run repayment. In this situation, the attacker would always be obtaining profits since `totalRedeemable` value is increased after every repayment.

Proof of Concept

```
function increaseTotalSupply(uint256 _amount) private {
    daiMock.mint(address(this), _amount);
    daiMock.approve(address(uToken), _amount);
    uToken.mint(_amount);
}

function testMintRedeemSandwich() public {
    increaseTotalSupply(50 ether);

    vm.prank(ALICE);
    uToken.borrow(ALICE, 50 ether);
    uint256 borrowed = uToken.borrowBalanceView(ALICE);

    vm.roll(block.number + 500);

    vm.startPrank(BOB);
    daiMock.approve(address(uToken), 100 ether);
    uToken.mint(100 ether);

    console.log("\n  [UToken] Total supply:", uToken.totalSupply());
```

```
    console.log("[UToken] BOB balance:", uToken.balanceOf(BOB));
    console.log("[DAI]    BOB balance:", daiMock.balanceOf(BOB));

    uint256 currExchangeRate = uToken.exchangeRateStored();
    console.log("[1] Exchange rate:", currExchangeRate);
    vm.stopPrank();

    vm.startPrank(ALICE);
    uint256 interest = uToken.calculatingInterest(ALICE);
    uint256 repayAmount = borrowed + interest;

    daiMock.approve(address(uToken), repayAmount);
    uToken.repayBorrow(ALICE, repayAmount);

    console.log("\n  [UToken] Total supply:", uToken.totalSupply());
    console.log("[UToken] ALICE balance:", uToken.balanceOf(ALICE));
    console.log("[DAI]    ALICE balance:", daiMock.balanceOf(ALICE));

    currExchangeRate = uToken.exchangeRateStored();
    console.log("[2] Exchange rate:", currExchangeRate);
    vm.stopPrank();

    vm.startPrank(BOB);
    uToken.redeem(uToken.balanceOf(BOB), 0);

    console.log("\n  [UToken] Total supply:", uToken.totalSupply());
    console.log("[UToken] BOB balance:", uToken.balanceOf(BOB));
    console.log("[DAI]    BOB balance:", daiMock.balanceOf(BOB));

    currExchangeRate = uToken.exchangeRateStored();
    console.log("[3] Exchange rate:", currExchangeRate);
}
```

Result

```
[PASS] testMintRedeemSandwich() (gas: 560119)
Logs:

  [UToken] Total supply: 150000000000000000000
  [UToken] BOB balance: 100000000000000000000
  [DAI]    BOB balance: 0
  [1] Exchange rate: 1000000000000000000

  [UToken] Total supply: 150000000000000000000
  [UToken] ALICE balance: 0
  [DAI]    ALICE balance: 99474750000000000000
```

```
  [2] Exchange rate: 1000084166666666666

  [UToken] Total supply: 50000000000000000000
  [UToken] BOB balance: 0
  [DAI]    BOB balance: 100008416666666666600
  [3] Exchange rate: 1000084166666666668
```

Impact

An attacker could always get profits from front-running repayments by taking advantage of `exchangeRateStored()` calculation before a repayment is made.

Code Snippet

exchangeRateStored()

```
function exchangeRateStored() public view returns (uint256) {
    uint256 totalSupply_ = totalSupply();
    return totalSupply_ == 0 ? initialExchangeRateMantissa : (totalRedeemable *
↪   WAD) / totalSupply_;
}
```

_repayBorrowFresh()

```
function _repayBorrowFresh(address payer, address borrower, uint256 amount,
↪   uint256 interest) internal {
    if (getBlockNumber() != accrualBlockNumber) revert AccrueBlockParity();
    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;

        // 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, (repayAmount -
↪   interest).toUint96(), 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).onRepayBorrow(borrower);
        }

        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();
        }
    } else {
        // For repayments that don't pay off the minimum we just need to adjust
↪   the
        // global balances and reduce the amount of interest accrued for the
↪   borrower
        toReserveAmount = (repayAmount * reserveFactorMantissa) / WAD;
        toRedeemableAmount = repayAmount - toReserveAmount;
        accountBorrows[borrower].interest = interest - repayAmount;
    }

    totalReserves += toReserveAmount;
    totalRedeemable += toRedeemableAmount;

    accountBorrows[borrower].interestIndex = borrowIndex;
```


```
    // Transfer underlying token that have been repaid and then deposit
    // then in the asset manager so they can be distributed between the
    // underlying money markets
    IERC20Upgradeable(underlying).safeTransferFrom(payer, address(this),
↪   repayAmount);
    _depositToAssetManager(repayAmount);

    emit LogRepay(payer, borrower, repayAmount);
}
```

Usageinmint()function

Usageinredeem()function

Tool used

Manual Review

Recommendation

An approach could be implementing TWAP in order to make front-running unprofitable in this situation.

Discussion

kingjacob

Udai minters dont earn interest because reserve factor is always set to 100% and on any UnionDAO network udai minters should not expect a yield. But this is a valid exploit if someone set reserve factor to a high amount.

the solution isnt twap or anything duration based because you cant allocate unsecured interest before its paid. On a short time horizon this leads to some minters getting more than those whove been in the pool longer, But over a long enough horizon it averages out. So the solution is an exit fee paid to reservefactor to make 1-3 block deposits and anything flashloan enabled unprofitable.

Issue M-6: Index mismatch on vouchers and vouchees might lead to invalid borrower-voucher-staker association

Source: https://github.com/sherlock-audit/2023-02-union-judging/issues/4

Found by

weeeh_

Summary

In certain situation, we can have a wrong borrower-voucher-staker association on the smart contract `UserManager.sol`

Vulnerability Detail

When `vouchers[borrower]` does contain only one element, which means borrower does only have one vouche, and the staker gave more than one vouche to borrowers, and so `vouchees[staker].length > 1` and `vouchees[staker][0].borrower != borrower`. Then as shown on loc https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L604 the `vouchees[staker][0].voucherIndex` will be overwritten to 0, which might be the wrong value, because we are modifying the struct Vouchee associated to another borrower.

Impact

The issue might impact the integrity of the contract by mismatching borrowers and stakers

Code Snippet

https://github.com/sherlock-audit/2023-02-union/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L583-L607

Tool used

vim Manual Review

Recommendation

We suggest to check the `vouchers[borrower]` length before writing to `vouchees` map.