SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Union Finance
Prepared by:	Sherlock
Lead Security Expert:	hyh
Dates Audited:	July 8 - July 13, 2024
Prepared on:	September 17, 2024

Introduction

Union is a p2p credit network protocol for permissionless underwriting and borrowing that adds support for arbitrary decimal tokens.

Scope

Repository: unioncredit/union-v2-contracts

Branch: feature/uni-1825-deploy-to-base-sepolia-testnet

Commit: 0ce70366459995bd5dd0102f8159ea628049214e

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
5	4

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

Varun_05
Bigsam
hyh
trachev
twicek
KungFuPanda
cryptphi
0xAadi
korok

Bugvorus
snapishere
MohammedRizwan

tsueti_
JuggerNaut63
Bauchibred

smbv-1923
bareli
0xmystery

Issue H-1: Wrong calculation of Accure Reward in Comptroller.sol

Source: https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/26

Found by

Bigsam, Varun_05

Summary

Accure Reward calculates the global total staked before updating the total amount of token frozen hence, the globalTotalStaked used does not reflect the actual present globalTotalStaked value. Thus A user can claim the reward for recently frozen tokens due to this error.

Vulnerability Detail

Based on the calculation in Comptroller.sol and UserManager,

function '_accrueReward' calculates the TotalStaked but gets the value of newly frozen totals and increments the _totalfrozen. This allows us to pass in a Larger amount of globalTotalStaked than we should at the current time, here the user benefits from this since the globalTotalStaked is used to get the amount(REWARD) and calculate the glnflationIndex.

```
 function _accrueRewards(address account, address token) private returns
↪  (uint256) {
        IUserManager userManager = _getUserManager(token);

@audit>>issue>>         // Lookup global state from UserManager
                                    uint256 globalTotalStaked =
↪  userManager.globalTotalStaked();

        // Lookup account state from UserManager
        UserManagerAccountState memory user = UserManagerAccountState(0, 0,
↪  false);
  @audit>>issue>>           (user.effectiveStaked, user.effectiveLocked,
↪  user.isMember) = userManager.onWithdrawRewards(account);

        uint256 amount = _calculateRewardsInternal(account, token,
↪  globalTotalStaked, user);

        // update the global states
```

```
        gInflationIndex = _getInflationIndexNew(globalTotalStaked,
↪  getTimestamp() - gLastUpdated);
        gLastUpdated = getTimestamp();
        users[account][token].inflationIndex = gInflationIndex;

        return amount;
    }
```

For Reference, please look at the same implementation when it is called directly from Usermanager.sol in function 'batchUpdateFrozenInfo'.

```
function batchUpdateFrozenInfo(address[] calldata stakerList) external
↪  whenNotPaused {
       uint256 stakerLength = stakerList.length;
    ------------------------------------------------------------------------
  @audit >>   update is done first >>      (, , uint256 memberTotalFrozen) =
↪  _getEffectiveAmounts(staker);

           uint256 memberFrozenBefore = _memberFrozen[staker];
           if (memberFrozenBefore != memberTotalFrozen) {
               _memberFrozen[staker] = memberTotalFrozen;
    @audit >>  totalfrozen is obtained>>           tmpTotalFrozen =
↪  tmpTotalFrozen - memberFrozenBefore + memberTotalFrozen;
           }
       }
       _totalFrozen = tmpTotalFrozen;

           @audit >> totalstaked - the present totalfrozen>>
↪  comptroller.updateTotalStaked(stakingToken, _totalStaked - _totalFrozen);
   }
```

The present globalTotalStaked is correctly implement in another implementation (globalTotal = _totalStaked - _totalFrozen;)

```
function updateTotalStaked(
       address token,
       uint256 totalStaked
   ) external override whenNotPaused onlyUserManager(token) returns (bool) {
       if (totalStaked > 0) {
           gInflationIndex = _getInflationIndexNew(totalStaked, getTimestamp() -
↪  gLastUpdated);
           gLastUpdated = getTimestamp();
       }

       return true;
```

```
    }
```

From the second implementation by the protocol, it should be noted that globalTotalStaked should only be called after the _totalfrozen has been correctly updated and not before.

Impact

Collection of Reward on newly frozen Tokens.

Code Snippet

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L224

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L1124-L1125

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/token/Comptroller.sol#L228

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L1056

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/user/UserManager.sol#L1067

Tool used

Manual Review

Recommendation

Get 'globalTotalStaked' after calculating all frozen assets in the userManager for the user.

```
    --  // Lookup global state from UserManager
    --  uint256 globalTotalStaked = userManager.globalTotalStaked();

       // Lookup account state from UserManager
      UserManagerAccountState memory user = UserManagerAccountState(0, 0, false);
      (user.effectiveStaked, user.effectiveLocked, user.isMember) =
↪   userManager.onWithdrawRewards(account);
```

```
++   // Lookup global state from UserManager
++  uint256 globalTotalStaked = userManager.globalTotalStaked();
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/unioncredit/union-v2-contracts/pull/180

dmitriia

> The protocol team fixed this issue in the following PRs/commits:
> unioncredit/union-v2-contracts#180

Fix looks ok

Issue H-2: VouchFaucet can be immediately drained by anyone

Source: https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/33

The protocol has acknowledged this issue.

Found by

0xAadi, Bigsam, Bugvorus, Varun_05, cryptphi, korok, snapishere, trachev

Summary

The `claimTokens` function in the VouchFaucet contract fails to properly enforce the `maxClaimable` limit because it does not update the value in the `claimedTokens` mapping. This allows any address to claim an arbitrary amount of any token, potentially draining the entire token balance of the contract, in a single transaction or through multiple transactions.

Vulnerability Detail

Included below is the relevant code from the VouchFaucet followed by key insights:

```
/// @notice Token address to msg sender to claimed amount
mapping(address => mapping(address => uint256)) public claimedTokens;

/// @notice Token address to max claimable amount
mapping(address => uint256) public maxClaimable;

/// @notice Claim tokens from this contract
function claimTokens(address token, uint256 amount) external {
    require(claimedTokens[token][msg.sender] <= maxClaimable[token],
↪   "amount>max");
    IERC20(token).transfer(msg.sender, amount);
    emit TokensClaimed(msg.sender, token, amount);
}

/// @notice Transfer ERC20 tokens
function transferERC20(address token, address to, uint256 amount) external
↪   onlyOwner {
    IERC20(token).transfer(to, amount);
}
```

- The claimedTokens mapping is never updated. It will always return 0 when looking up how much of any token has been claimed by any address.
- The maxClaimable mapping will by default return 0 for any token the contract could ever receive. The contract owner can use the setMaxClaimable function but is only able to set the claimable amount for any token to 0 or greater.
- The transferERC20 function is protected by the onlyOwner modifier signaling the desire to restrict access to this type of transfer.
- Currently no matter what the admin does the require statement in claimTokens will always pass because it evaluates an expression that will always effectively be: `require(0 <= [uint256], "amount>max");` This makes claimTokens effectively equivalent to an unrestricted version of transferERC20.

Proof of Concept

The proof of concept below imports and utilizes the protcols own TestWrapper for simplicity in setting up a realistic testing environment.

The test case demonstrates that despite the VouchFaucet containing mechanisms that clearly intend to disallow the faucet from being easily drained by a single address, such an outcome is possible with no effort.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.16;

import {Test, console} from "forge-std/Test.sol";
import {TestWrapper} from "../TestWrapper.sol";
import {VouchFaucet} from "../../src/contracts/peripheral/VouchFaucet.sol";

import {IERC20} from "@openzeppelin/token/ERC20/IERC20.sol";

contract TestVouchFaucet is TestWrapper {
    VouchFaucet public vouchFaucet;
    uint256 public TRUST_AMOUNT = 10 * UNIT;

    function setUp() public {
        deployMocks();
        vouchFaucet = new VouchFaucet(address(userManagerMock), TRUST_AMOUNT);
    }

    function testDrainVouchFaucet() public {
        address bob = address(1234);

        erc20Mock.mint(address(vouchFaucet), 3 * UNIT);
```

```
        vouchFaucet.setMaxClaimable(address(erc20Mock), 1 * UNIT);
        assertEq(vouchFaucet.maxClaimable(address(erc20Mock)), 1 * UNIT);

        // Bob can claim any number of tokens despite maxClaimable set to 1 Unit
        vm.prank(bob);
        vouchFaucet.claimTokens(address(erc20Mock), 3 * UNIT);

        assertEq(IERC20(erc20Mock).balanceOf(bob), 3 * UNIT);
    }

}
```

Impact

Without the intended enforcement provided by the require statement the claimTokens function provides unrestricted external access to an ERC20 transfer function. This is clearly not intended as demonstrated by the presence of the onlyOwner on the similar transferERC20 function. The result is that any caller can immediately transfer out any amount of any token.

This oversight completely undermines the token distribution model of the faucet. If deployed without modification it would render the contract useless for the intended purpose due to its inability to securely hold any amount of any token.

Code Snippet

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/peripheral/VouchFaucet.sol#L93-L97

Tool used

Manual Review

Recommendation

The following correction ensures that any individual address can't claim more than the set claimable amount, maintaining the intended token distribution model of the faucet.

```
File: VouchFaucet.sol

    function claimTokens(address token, uint256 amount) external {
```

```
-         require(claimedTokens[token][msg.sender] <= maxClaimable[token],
 ↪  "amount>max");
+         uint256 newTotal = claimedTokens[token][msg.sender] + amount;
+         require(newTotal <= maxClaimable[token], "Exceeds max claimable
 ↪  amount");

+         claimedTokens[token][msg.sender] = newTotal;
          IERC20(token).transfer(msg.sender, amount);

          emit TokensClaimed(msg.sender, token, amount);
      }
```

The following additional recommendations should be considered. The suggestions won't impact legitimate users, but raise the effort required for an malicious actor to disrupt the intended functioning of the contract.

1. Ensure the claimedTokens mapping is updated before the transfer to avoid reentrancy risk, OpenZeppelin ReentrancyGuard could also be considered.
2. Consider adding an admin adjustable global cap on total tokens that can be claimed across all addresses to help control faucet outflows.
3. Consider implementing a time-based cooldown mechanism to limit the frequency of claims per address.

Discussion

c-plus-plus-equals-c-plus-one

This highlights the exactly same problem as https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/96 does. So these two issues should probably be merged as one single, duplicates of each other.

c-plus-plus-equals-c-plus-one

@WangSecurity is this one escalated too just like #96 is?

Issue H-3: Repaying a Loan with Permit in UErc20.sol Wrongly calculates the interest to be paid this Reduce/Increase profits for the protocol as interest calculations are not performed correctly.

Source: https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/43

Found by

Bigsam, Varun_05, hyh, trachev, twicek

Summary

The function `_repayBorrowFresh` in `UErc20.sol` wrongly calculates the interest to be paid when repaying a loan with a permit. The issue arises because the amount to repay is scaled to 1e18, but the interest is not, leading to users passing incorrect interest amounts based on the token's decimal places. This can result in users paying significantly less interest than they should for tokens with fewer decimal places (e.g., USDC, USDT) or more than they should for tokens with more decimal places.

Vulnerability Detail

The core of the problem lies in the different scaling applied to the amount and interest during the repayment process. In `UErc20.sol`, the function `_repayBorrowFresh` is called with the amount scaled to 1e18 but the interest remains unscaled, which causes incorrect interest calculations.

```
function repayBorrowWithERC20Permit(
       address borrower,
       uint256 amount,
       uint256 deadline,
       uint8 v,
       bytes32 r,
       bytes32 s
   ) external whenNotPaused {
       IERC20Permit erc20Token = IERC20Permit(underlying);
       erc20Token.permit(msg.sender, address(this), amount, deadline, v, r, s);

       if (!accrueInterest()) revert AccrueInterestFailed();
 @audit>> Wrong interest passed>> notscaled>>      uint256 interest =
↪   calculatingInterest(borrower);
```

```
        _repayBorrowFresh(msg.sender, borrower, decimalScaling(amount,
↪  underlyingDecimal), interest);
    }
```

Example

For tokens like USDC or USDT (which have 6 decimal places):

- If the interest to be paid is $2,000,000, the expected interest should be scaled to 1e18, resulting in $2,000,000 * 1e18.
- However, since the interest is not scaled, it remains at $2,000,000 * 1e6, leading to the contract recording less interest than should, the excess amount remains in the pool and it is not shared accordingly.

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/market/UToken.sol#L568-L570 we use the above instead of using

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/market/UToken.sol#L682

For less amount of interest used in calculation this funds are stuck in the contract since it is not assigned to anyone to claim.
https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/market/UToken.sol#L705-L710

For tokens with more than 18 decimal places, the user would end up paying more interest than they should to the protocol and less funds will be available for all the lenders to claim their funds, resulting in overpayment.

This discrepancy creates an opportunity for the Wrong sharing of interest shares by protocol.

Impact

The impact of this vulnerability includes:

- Users potentially paying incorrect interest amounts, leading to financial losses for the protocol.
- Reduced profits for the protocol as interest calculations are not performed correctly.

Here is the problematic code snippet from `UErc20.sol`:

```
uint256 interest = calculatingInterest(borrower);
```

```
_repayBorrowFresh(msg.sender, borrower, decimalScaling(amount,
↪  underlyingDecimal), interest);
```

In `UDai.sol`, although it is less critical because DAI has 18 decimals, the same logic flaw exists.

Below is the Reference to the code if the user pays normally

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/market/UToken.sol#L679-L684

Interest is scaled there properly and sent to the _repayBorrowFresh and use the permit because if the incorrect implementation.

Code Snippet

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/market/UErc20.sol#L8-L22

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/market/UErc20.sol#L19-L21

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/market/UToken.sol#L705-L710

This issue is absent in Dai because it decimal is 1e18
https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/market/UDai.sol#L9-L23

Tool used

Manual Review

Recommendation

To address this issue, the interest should be scaled correctly according to the token's decimal places. The function call should ensure both the amount and the interest are appropriately scaled. Here's the recommended fix:

```
function repayBorrowWithPermit(uint256 amount, ...) external {

 ..........................................................................
↪  ............
    --       uint256 interest = calculatingInterest(borrower);
    ++     uint256 interest = _calculatingInterest(borrower);
```

```
            _repayBorrowFresh(msg.sender, borrower, decimalScaling(amount,
↪  underlyingDecimal), Interest);
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/unioncredit/union-v2-contracts/pull/177

dmitriia

> The protocol team fixed this issue in the following PRs/commits:
> unioncredit/union-v2-contracts#177

Fix looks ok

Issue H-4: The _totalStaked tracker calculation is incorrect and will be inflated due to the improper logic in the writeOffDebt function of the UserManager contract, leading to wrong Comptroller gInflationIndex being calculated and wrong user rewards being issued

Source: https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/105

Found by

KungFuPanda, Varun_05, hyh, trachev, twicek

Summary

During `debtWriteOff` call in the `UserManager`, subtracting the `amount` instead of `realAmount` will lead to the whole `gInflationIndex` being inflated in the `Comptroller` contract, as well as general accounting inflation.

Due to the `UserManager`'s `_totalStaked` being coupled with the `Comptroller`'s `gInflationIndex`, the `userInfo` stake's `inflationIndex` will be calculated absolutely incorrectly:

```
/**
 *  @dev Calculate currently unclaimed rewards
 *  @param account Account address
 *  @param token Staking token address
 *  @param totalStaked Effective total staked
 *  @param user User account global state
 *  @return Unclaimed rewards
 */
function _calculateRewardsInternal(
    address account,
    address token,
    uint256 totalStaked,
    UserManagerAccountState memory user
) internal view returns (uint256) {
    Info memory userInfo = users[account][token];
    uint256 startInflationIndex = userInfo.inflationIndex; // @@ <<< this
 ↪  internally depends on the UserManager's _totalStaked variable
```

And due to that, the whole user rewards calculation will be incorrect:

```
    uint256 rewardMultiplier = _getRewardsMultiplier(user);

    uint256 curInflationIndex = _getInflationIndexNew(totalStaked,
↪   getTimestamp() - gLastUpdated);

    if (curInflationIndex < startInflationIndex) revert InflationIndexTooSmall();

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

(https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/token/Comptroller.sol#L293)

Furthermore, during the reward accrual, there's another outcome of the miscalculated `_totalStaked` and `gInflationIndex` values:

```
function _accrueRewards(address account, address token) private returns
↪   (uint256) {
    IUserManager userManager = _getUserManager(token);

    // Lookup global state from UserManager
    uint256 globalTotalStaked = userManager.globalTotalStaked(); // @@ <<< here
↪   a wrong _totalStaked amount is retrieved!!!

    // Lookup account state from UserManager
    UserManagerAccountState memory user = UserManagerAccountState(0, 0, false);
    (user.effectiveStaked, user.effectiveLocked, user.isMember) =
↪   userManager.onWithdrawRewards(account);

    uint256 amount = _calculateRewardsInternal(account, token,
↪   globalTotalStaked, user); // @@ <<< here a wrong amount is passed down to
↪   the calculation!

    // update the global states
    gInflationIndex = _getInflationIndexNew(globalTotalStaked, getTimestamp() -
↪   gLastUpdated);
    gLastUpdated = getTimestamp();
    users[account][token].inflationIndex = gInflationIndex;

    return amount;
```

```
}
```

Above is a reference from the `Comptroller` contract: https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/token/Comptroller.sol#L220C1-L238C6.

References for the main problem:

- https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/token/Comptroller.sol#L276
- https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/token/Comptroller.sol#L287

The culprit's details are further explained below.

This is due to using a non-scaled `amount` instead of the *scaled* `realAmount` in the `debtWriteOff` function here, in this line: https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/user/UserManager.sol#L834

Root Cause

In this update Union Finance adds distinct definitions for the `actualAmount` *being the **real** scaled token amount*, and the `amount` *being an unscaled nominal amount* that is to be scaled by `stakingTokenDecimal`, and is usually accepted as an argument for functions within the `UserManager` contract.

Both `stake` and `unstake` functions track the `totalStaked` balance as a **scaled** AND **real** token amount, as can be seen here: https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/user/UserManager.sol#L784. The snippet:

```
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 < decimalScaling(amount,
↪   stakingTokenDecimal))
        revert InsufficientBalance();

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

```
    uint256 remaining = IAssetManager(assetManager).withdraw(stakingToken,
↪   msg.sender, amount);
    if (remaining > amount) {
        revert AssetManagerWithdrawFailed();
    }
    uint96 actualAmount = decimalScaling(uint256(amount) - remaining,
↪   stakingTokenDecimal).toUint96();

    staker.stakedAmount -= actualAmount;
    _totalStaked -= actualAmount; // @@ <<< the actualAmount is subtracted

    emit LogUnstake(msg.sender, amount - remaining.toUint96());
}
```

And for `stake` it's `actualAmount` too (https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/user/UserManager.sol#L748), correspondingly:

```
function stake(uint96 amount) public whenNotPaused nonReentrant {
    IERC20Upgradeable erc20Token = IERC20Upgradeable(stakingToken);
    uint96 actualAmount = decimalScaling(uint256(amount),
↪   stakingTokenDecimal).toUint96();
    comptroller.withdrawRewards(msg.sender, stakingToken);

    Staker storage staker = _stakers[msg.sender];

    if (staker.stakedAmount + actualAmount > _maxStakeAmount) revert
↪   StakeLimitReached();

    staker.stakedAmount += actualAmount;
    _totalStaked += actualAmount; // @@ <<< here you can see it!

    erc20Token.safeTransferFrom(msg.sender, address(this), amount);
    uint256 currentAllowance = erc20Token.allowance(address(this), assetManager);
    if (currentAllowance < amount) {
        erc20Token.safeIncreaseAllowance(assetManager, amount -
↪   currentAllowance);
    }

    if (!IAssetManager(assetManager).deposit(stakingToken, amount)) revert
↪   AssetManagerDepositFailed();
    emit LogStake(msg.sender, amount);
}
```

However, the `debtWriteOff` function doesn't subtract the `actualAmount`, but

decreases the `_totalStaked` counter by a non-scaled `amount` value:

```
Staker storage staker = _stakers[stakerAddress];

staker.stakedAmount -= actualAmount.toUint96();
staker.locked -= actualAmount.toUint96();
staker.lastUpdated = currTime.toUint64();

_totalStaked -= amount;

// update vouch trust amount
vouch.trust -= actualAmount.toUint96();
vouch.locked -= actualAmount.toUint96();
vouch.lastUpdated = currTime.toUint64();
```

Then here later in the `batchUpdateFrozenInfo` function (that can be called by anyone and is unrestricted!), the `Comptroller` contract is notified of the new `_totalStaked` amount (https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/user/UserManager.sol#L1121):

```
    comptroller.updateTotalStaked(stakingToken, _totalStaked - _totalFrozen);
}

function globalTotalStaked() external view returns (uint256 globalTotal) {
    globalTotal = _totalStaked - _totalFrozen;
}
```

The problem lies in this change here: https://github.com/unioncredit/union-v2-contracts/pull/172/files#diff-e274f419b6384471f87c2b7d6a2c75150b95d37f3174a21d7d675ad20e3e4464R834

The `Comptroller`'s `updateTotalStaked` function gets called:

```
/**
 *  @dev When total staked change update inflation index
 *  @param totalStaked totalStaked amount
 *  @return Whether succeeded
 */
function updateTotalStaked(
    address token,
    uint256 totalStaked
) external override whenNotPaused onlyUserManager(token) returns (bool) {
    if (totalStaked > 0) {
        gInflationIndex = _getInflationIndexNew(totalStaked, getTimestamp() -
↪   gLastUpdated);
```

```
            gLastUpdated = getTimestamp();
        }

        return true;
    }
```

And finally, the `gInflationIndex` value will be inflated.

Internal pre-conditions

1. As far as I can tell, the attack will be unintentional in most cases, happening automatically on each `debtWriteOff` call, because the culprit is an improper calculation.
2. Or this can be utilized together with calling `batchUpdateFrozenInfo` to inflate the `gInflationIndex` value intentionally and cause the `Comptroller`'s `_getInflationIndexNew` to return incorrect results:

```
gInflationIndex = _getInflationIndexNew(totalStaked, getTimestamp() -
↪   gLastUpdated);
```

External pre-conditions

None. The bug is just implicitly there.

Attack Path

As it's a mistake in the `_totalStaked` calculation logic, there's no particular trigger for this attack, as it will happen if any user `write`'s`OffDebt`.

Impact

The whole `gInflationIndex` will be inflated, and will be calculated incorrectly.

Besides that, tracking the wrong amount of the currently active staked tokens will be misleading for the external users that refer to that value.

PoC

None. Please leave me a comment if you request one from me.

Mitigation

Instead of subtracting `amount`, you should subtract the `actualAmount` from the `_totalStaked` variable here in `writeOffDebt`:

```
        Staker storage staker = _stakers[stakerAddress];

        staker.stakedAmount -= actualAmount.toUint96();
        staker.locked -= actualAmount.toUint96();
        staker.lastUpdated = currTime.toUint64();

-       _totalStaked -= amount;
+       _totalStaked -= actualAmount;

        // update vouch trust amount
        vouch.trust -= actualAmount.toUint96();
        vouch.locked -= actualAmount.toUint96();
        vouch.lastUpdated = currTime.toUint64();
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/unioncredit/union-v2-contracts/pull/179

dmitriia

> The protocol team fixed this issue in the following PRs/commits:
> unioncredit/union-v2-contracts#179

Fix looks ok

Issue M-1: Possible loss of funds, transfer functions can silently fail

Source: https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/22

The protocol has acknowledged this issue.

Found by

Bauchibred, JuggerNaut63, MohammedRizwan, smbv-1923, tsueti_

Summary

Possible loss of funds, transfer functions can silently fail

Vulnerability Detail

`Union` Protocol's contracts are expected to be used USDT, USDC and DAI. The contracts will be deployed on Any EVM compatible chain which also includes Ethereum mainnet itself. Both of these details are mentioned in contest readme. This issue is specifically for tokens like USDT and similar tokens etc on Ethereum mainnet.

The following functions makes use of ERC20's `transferFrom()` in following contracts:

1) In `VouchFaucet.sol`, the `claimTokens()` is called by users to claim the tokens and the token is transferred to user but the transfer return value is not checked and similarly in case of `transferERC20()` function.

```
    function claimTokens(address token, uint256 amount) external {
        require(claimedTokens[token][msg.sender] <= maxClaimable[token],
↪   "amount>max");
@>          IERC20(token).transfer(msg.sender, amount);      @audit // unchecked
↪   transfer return value
        emit TokensClaimed(msg.sender, token, amount);
    }


    function transferERC20(address token, address to, uint256 amount) external
↪   onlyOwner {
@>          IERC20(token).transfer(to, amount);                 @audit // unchecked
↪   transfer return value
    }
```

2) In `ERC1155Voucher.transferERC20()`, tokens are being transferred to recipient address and return value is not checked.

```
    function transferERC20(address token, address to, uint256 amount) external
↪   onlyOwner {
@>          IERC20(token).transfer(to, amount);            @audit // unchecked
↪   transfer return value
    }
```

The issue here is with the use of unsafe `transfer()` function. The `ERC20.transfer()` function return a boolean value indicating success. This parameter needs to be checked for success. Some tokens do not revert if the transfer failed but return false instead.

Some tokens like `USDT` don't correctly implement the EIP20 standard and their transfer() function return void instead of a success boolean. Calling these functions with the correct EIP20 function signatures will always revert.

Tokens that don't actually perform the transfer and return false are still counted as a correct transfer and tokens that don't correctly implement the latest EIP20 spec, like USDT, will be unusable in the protocol as they revert the transaction because of the missing return value. There could be silent failure in transfer which may lead to loss of user funds in `ERC1155Voucher.transferERC20()` and `VouchFaucet.claimTokens()`

Impact

Tokens that don't actually perform the transfer and return false are still counted as a correct transfer and tokens that don't correctly implement the latest EIP20 spec will be unusable in the protocol as they revert the transaction because of the missing return value. This will lead to loss of user funds.

Code Snippet

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/peripheral/VouchFaucet.sol#L95

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/peripheral/VouchFaucet.sol#L124

Tool used

Manual Review

Recommendation

Use OpenZeppelin's SafeERC20 versions with the `safeTransfer()` function instead of `transfer()`.

For example, consider below changes in `VouchFaucet.sol`:

```
+ import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract VouchFaucet is Ownable {

+       using SafeERC20 for IERC20;

    function claimTokens(address token, uint256 amount) external {
        require(claimedTokens[token][msg.sender] <= maxClaimable[token],
↪   "amount>max");
-         IERC20(token).transfer(msg.sender, amount);
+        IERC20(token).safeTransfer(msg.sender, amount);
        emit TokensClaimed(msg.sender, token, amount);
    }


    function transferERC20(address token, address to, uint256 amount) external
↪   onlyOwner {
-         IERC20(token).transfer(to, amount);
+         IERC20(token).safeTransfer(to, amount);
    }
```

Discussion

sherlock-admin3

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

0xmystery commented:

> Low QA on safeTransfer

0xRizwan

I believe, this issue is incorrectly judged.

@mystery0x

> Low QA on safeTransfer

There is no such rule at sherlock.

Per the contest readme, the protocol is expected to be deployed on any EVM compatible network so this includes `Ethereum` mainnet itself and the above issue

regarding safe transfers is relevant for Ethereum mainnet only.

> On what chains are the smart contracts going to be deployed? Any EVM compatible network

contest readme further confirms, USDT/USDC would be used in contracts.

> If you are integrating tokens, are you allowing only whitelisted tokens to work with the codebase or any complying with the standard? Are they assumed to have certain properties, e.g. be non-reentrant? Are there any types of weird tokens you want to integrate? USDC, USDT, DAI

Per sherlock rules,

> 23. Non-Standard tokens: Issues related to tokens with non-standard behaviors, such as weird-tokens are not considered valid by default unless these tokens are explicitly mentioned in the README.

Since, USDT is mentioned in readme so the non-standard behabiour pertaining to USDT would be valid issue for this contest.

Affected functions would be `transferERC20()` in `VouchFaucet.sol` and `ERC1155Voucher.sol`.

```
function transferERC20(address token, address to, uint256 amount) external
↪   onlyOwner {
    IERC20(token).transfer(to, amount);
}
```

Similar such issue with similar readme questionare had been judged as Medium severity. see- Notional issue and Teller

Therefore, based on above arguements this issue should be valid medium severity.

0xMR0

Escalate

on behalf of @0xRizwan

sherlock-admin3

> Escalate
>
> on behalf of @0xRizwan

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

mystery0x

This is a widely known issue typically deemed low/informational. Will let the sponsors and the Sherlock judge decide its medium validity.

WangSecurity

Just checking for transfer return value is invalid I agree, but the problem here is that regular `transfer` will try to extract a boolean from the transfer which will revert, so for example, the users won't be able to call claimTokens in VouchFacet. But, regular `transfer` is used only in VouchFacet and `ERC1155Voucher`, @mystery0x and @0xRizwan could you assist me with how severe these would be the revert in these functions, and what operations would be impossible in case of such reverts?

Bauchibred

HI @WangSecurity, to chime in here, wouldn't the fact that these contracts do not work with USDT suffice as medium, since it's been indicated in the readMe that the protocol is to work with USDT, USDC & DAI? If yes, do check #17, I was initially waiting till the resolution of this escalation before hinting it's a duplicate. But I believe both reports are duplicates and #17 does show how `ERC1155Voucher#transferERC20()` & `VouchFaucet#transferERC20()` would never work with USDT.

WangSecurity

@Bauchibred fair point, still I wanted to know if maybe there's no impact and these functions wouldn't be used. But further considering this, I agree it's medium severity, since regardless of the goal of these functions, they won't work with USDT. Planning to accept the escalation. The duplicate is #17, @mystery0x @Bauchibred @0xRizwan are there any other duplicates?

MD-YashShah1923

@WangSecurity https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/116 this as well is duplicate of this issue.

0xRizwan

@WangSecurity #11 #17 #57 #116 should be duplicates.

WangSecurity

Result: Medium Has duplicates

PS: I see some of the duplicates are insufficient, I'll comment on them directly explaining why.

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0xMR0: accepted

Issue M-2: `ERC1155Voucher.onERC1155BatchReceived()` does not check the caller is the valid token therefore any un-registered token can invoke `onERC1155BatchReceived()`

Source: https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/23

The protocol has acknowledged this issue.

Found by

0xAadi, 0xmystery, KungFuPanda, MohammedRizwan, bareli, cryptphi, korok, trachev

Summary

`ERC1155Voucher.onERC1155BatchReceived()` does not check the caller is the valid token therefore any unregistered token can invoke `onERC1155BatchReceived()`

Vulnerability Detail

`ERC1155Voucher.sol` is the voucher contract that takes `ERC1155` tokens as deposits and gives a vouch. An ERC1155 token can invoke two safe methods:

1) `onERC1155Received()` and
2) `onERC1155BatchReceived()`

An ERC1155-compliant smart contract must call above functions on the token recipient contract, at the end of a `safeTransferFrom` and `safeBatchTransferFrom` respectively, after the balance has been updated.

The `ERC1155Voucher` contract owner can set the valid token i.e ERC1155 token which can invoke both `onERC1155Received()` and `onERC1155BatchReceived()` functions.

```
mapping(address => bool) public isValidToken;


function setIsValid(address token, bool isValid) external onlyOwner {
    isValidToken[token] = isValid;
    emit SetIsValidToken(token, isValid);
}
```

The valid token i.e msg.sender calling the `onERC1155Received()` is checked in `ERC1155Voucher.onERC1155Received()` function

```
    function onERC1155Received(
        address operator,
        address from,
        uint256 id,
        uint256 value,
        bytes calldata data
    ) external returns (bytes4) {
@>          require(isValidToken[msg.sender], "!valid token");
        _vouchFor(from);
        return bytes4(keccak256("onERC1155Received(address,address,uint256,uint2
↪   56,bytes)"));
    }
```

This means that only the valid tokens set by contract owner can invoke the `ERC1155Voucher.onERC1155Received()` function. However, this particular check is missing in `ERC1155Voucher.onERC1155BatchReceived()` function.

```
function onERC1155BatchReceived(
    address operator,
    address from,
    uint256[] calldata ids,
    uint256[] calldata values,
    bytes calldata data
) external returns (bytes4) {
    _vouchFor(from);
    return bytes4(keccak256("onERC1155BatchReceived(address,address,uint256[],ui
↪   nt256[],bytes)"));
}
```

`onERC1155BatchReceived()` does not check the `isValidToken[msg.sender]` which means any ERC1155 token can call `ERC1155Voucher.onERC1155BatchReceived()` to deposit the ERC1155 to receive the vouch. This is not intended behaviour by protocol and would break the intended design of setting valid tokens by contract owner. Any in-valid tokens can easily call `onERC1155BatchReceived()` and can bypass the check at L-109 implemented in `onERC1155Received()` function.

Impact

Any in-valid or unregistered ERC1155 token can invoke the `onERC1155BatchReceived()` function which would make the check at L-109 of `onERC1155Received()` useless as batch function would allow to deposit ERC1155 to receive the vouch therefore bypassing the L-109 check in `onERC1155Received()`. This would break the design of protocol as valid tokens as msg.sender are not checked in `onERC1155BatchReceived()`.

Code Snippet

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/peripheral/ERC1155Voucher.sol#L121

Tool used

Manual Review

Recommendation

Consider checking `isValidToken[msg.sender]` in `onERC1155BatchReceived()` to invoke it from registered valid token only.

Consider below changes:

```
    function onERC1155BatchReceived(
        address operator,
        address from,
        uint256[] calldata ids,
        uint256[] calldata values,
        bytes calldata data
    ) external returns (bytes4) {
+       require(isValidToken[msg.sender], "!valid token");
        _vouchFor(from);
        return bytes4(keccak256("onERC1155BatchReceived(address,address,uint256[
↪ ],uint256[],bytes)"));
    }
```

Issue M-3: Any user can claim an unlimited amount of vouch in `VouchFaucet.sol`

Source: https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/102

The protocol has acknowledged this issue.

Found by

trachev

Summary

Currently, there is no validation performed in `VouchFaucet.sol` when `claimVouch` is called. This is highly dangerous as any untrusted user can increase their vouch and perform malicious borrows, stealing from the contract's stake.

Vulnerability Detail

As we can see in the `claimVouch` function there is no validation performed and it can be called by any address:

```
function claimVouch() external {
        IUserManager(USER_MANAGER).updateTrust(msg.sender, uint96(TRUST_AMOUNT));
        emit VouchClaimed(msg.sender);
}
```

In addition to that, the maximum amount of trust that any user can claim - `TRUST_AMOUNT` can easily be bypassed by calling `claimVouch`, borrowing the entire `TRUST_AMOUNT` and after that calling `claimVouch` again.

Impact

Malicious borrows can be made by untrusted users and the maximum amount that can be vouched for a user can be bypassed, putting the contract's funds at risk of being stolen.

Code Snippet

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/peripheral/VouchFaucet.sol#L87-L90

Tool used

Manual Review

Recommendation

Only users approved by the owner should be able to call `claimVouch` and they should not be able to claim more trust than `TRUST_AMOUNT`.

Discussion

sherlock-admin4

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

0xmystery commented:

> Intended design. (borrower == staker) is checked to prevent self vaouching in updateTrust()

trachevgeorgi

Escalate This issue has been incorrectly excluded. The comment from @mystery0x is not connected to the issue as the problem is not related to whether the borrower is the same as the staker. The issue revolves around the fact that any user can claim more trust than the allowed limit (`TRUST_AMOUNT`), thus allowing any user to claim as much trust as they wish from the `VouchFaucet.sol` contract. As a result, a single user can block any other users from successfully calling the `claimVouch` function. This is definitely not the intended behaviour of the function as most likely only one user will claim all of the contract's trust.

sherlock-admin3

> Escalate This issue has been incorrectly excluded. The comment from @mystery0x is not connected to the issue as the problem is not related to whether the borrower is the same as the staker. The issue revolves around the fact that any user can claim more trust than the allowed limit (`TRUST_AMOUNT`), thus allowing any user to claim as much trust as they wish from the `VouchFaucet.sol` contract. As a result, a single user can block any other users from successfully calling the `claimVouch` function. This is definitely not the intended behaviour of the function as most likely only one user will claim all of the contract's trust.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

mystery0x

There is `onlyMember` visibility in `updateTrust()`:

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/user/UserManager.sol#L586

```
function updateTrust(address borrower, uint96 trustAmount) external
↪  onlyMember(msg.sender) whenNotPaused {
```

It's a permissioned call eventually, i.e. only member can vouch someone. Given the context of the design, it doesn't seem to introduce a threat unless VouchFaucet.sol is a member.

trachevgeorgi

@mystery0x With all due respect, I believe you do not understand the issue correctly. The VouchFaucet.sol contract is going to be staking in Union, like a member. Therefore, after it stakes borrowers are going to be able to borrow assets from the VoucheFaucet.sol contract, like they would from any other member. The only way to borrow from a member is if the lender (VouchFaucet.sol in the case) calls updateTrust with the specific borrower as the first parameter. Essentially, the borrower is vouching for / increasing their trust in a certain member. In the issue above any address can increase their own trust on behalf of VouchFaucet.sol. The issue is that they can give themselves too much trust, due to insufficient validation. As a result, they are going to be able to borrower the entire stake of VouchFaucet.sol instead of the allowed TRUST_AMOUNT.

WangSecurity

As I understand not every staker is necessarily a member, so @trachevgeorgi could you elaborate more on how VouchFacet can become a member for `updateTrust` to be called?

trachevgeorgi

@WangSecurity Yes, I can. There are two ways that any address can become a member. The first one is the `addMember onlyAdmin` function, which allows the admin to make any address a member for free. The second one is the `registerMember(address newMember)` function, which allows users to make other addresses members by paying a fee of Union Token: https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/user/UserManager.sol#L722-L728 The owner of the VouchFoucet.sol contract will most likely use the second option.

Furthermore, it is important to add that if the VouchFaucet.sol contract is not a member, it is not going to be able to function, so being a member is an expected

necessity.

WangSecurity

> Furthermore, it is important to add that if the VouchFaucet.sol contract is not a member, it is not going to be able to function, so being a member is an expected necessity.

Can you elaborate more about it and where you got this info from?

Sorry for the late reply, I'm trying to get info on VouchFacet from the sponsor.

trachevgeorgi

> > Furthermore, it is important to add that if the VouchFaucet.sol contract is not a member, it is not going to be able to function, so being a member is an expected necessity.
>
> Can you elaborate more about it and where you got this info from?
>
> Sorry for the late reply, I'm trying to get info on VouchFacet from the sponsor.

Hi, I believe that many users will be members as this is the only way to actually lend your money. Being a member cannot be a very restrictive role that only certain addresses trusted by the protocol can have access to. We understand that as any address can register themselves or others as members by just paying a fee of Union Token, so it is definitely intended for any address to be possible and not difficult to become a member. Regarding your question on where I get the information from, if you look at the `registerMember` function and the developer comments above it you can get all of the information necessary: https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/7ffe43f68a1b8e8de1dfd9de5a4d89c90fd6f710/union-v2-contracts/contracts/user/UserManager.sol#L715-L728

Also it is important to notice that in order to vouch for somebody (vouching essentially means that you are allowing a borrower to borrow from your funds) you need to be a member, therefore all lenders will need to be members.

The purpose of VouchFaucet is for it to stake in the protocol and after that for the contract to be able to vouch for other users so that the users will be able to borrow from the VouchFaucet's stake, for this to work it is expected for VouchFaucet to be a member.

trachevgeorgi

@WangSecurity To add to the comment above, my report essentially indicates the issue that any user can increase how much the `VouchFaucet.sol` contract, which must be a member, has vouched for them, even though they should only be allowed to get only `TRUST_AMOUNT` of trust out of `VouchFaucet.sol`. For example, if `VouchFaucet.sol` has a stake of `2 * TRUST_AMOUNT`, a single user should only be able

to get `1 * TRUST_AMOUNT`, and therefore be able to borrow only `1 * TRUST_AMOUNT`, but currently they can borrow the entire `2 * TRUST_AMOUNT`.

WangSecurity

Thank you, I understand why it's important for the users to be members, but I meant why it's important for VouchFacet contract to be a member, specifically. Excuse me for the confusion, is this information in the docs or you cam to this conclusion by just getting more context of the code base?

I see VouchFacet is also used to stake in the User Manager. And agree it would be strange that there's a function to give trust but this contract wouldn't be a member.

What makes me confused here is that on one of other escalations one Watson shared a screenshot from the sponsor saying VouchFacet::claimTokens is just to rescue tokens that were accidentally sent into the contract. So that makes me confused about the entire VouchFacet and why it's going to be used.

trachevgeorgi

> Thank you, I understand why it's important for the users to be members, but I meant why it's important for VouchFacet contract to be a member, specifically. Excuse me for the confusion, is this information in the docs or you cam to this conclusion by just getting more context of the code base?
>
> I see VouchFacet is also used to stake in the User Manager. And agree it would be strange that there's a function to give trust but this contract wouldn't be a member.
>
> What makes me confused here is that on one of other escalations one Watson shared a screenshot from the sponsor saying VouchFacet::claimTokens is just to rescue tokens that were accidentally sent into the contract. So that makes me confused about the entire VouchFacet and why it's going to be used.

@WangSecurity Hi! There are no documentations on this contract but I believe it is pretty clear what its purpose is just by looking at the code and the contract's name. Firstly, the name VouchFaucet indicates that it is another way for users to vouch for other users. They just stake their funds and allow others to claim a certain amount of trust in order to borrow from the stake. It is impossible for claimVouch to be used to rescue anything as its sole purpose is to allow addresses to claim an amount of trust. I have pointed two issues with the code the main one being able to bypass the TRUST_AMOUNT limit which is why I believe the finding should be valid.

maxweng

Hi, this contract is only used for testing purposes on Testnet. And `claimVouch()` is just to allow testers to easily get vouches even no other members do it for them.

WangSecurity

But as I understand this wasn't mentioned anywhere during the contest, correct?

Also, @trachevgeorgi as I understand, even if VouchFaucet is not a member, any existing member can give vouchers to VouchFaucet and then call `registerMemeber` to make VouchFaucet a member, correct? In that way, this bug would be possible, if we disregard the fact that it's not intended to be deployed. Also, is there a way to remove a member?

trachevgeorgi

> But as I understand this wasn't mentioned anywhere during the contest, correct?
>
> Also, @trachevgeorgi as I understand, even if VouchFaucet is not a member, any existing member can give vouchers to VouchFaucet and then call `registerMemeber` to make VouchFaucet a member, correct? In that way, this bug would be possible, if we disregard the fact that it's not intended to be deployed. Also, is there a way to remove a member?

@WangSecurity Yes, anyone can make VouchFaucet a member and just as you said it was never mentioned where the contract will be deployed therefore it is still in scope. Also I do not find a function that can remove an address from the member list.

WangSecurity

Then, I agree it has to be a valid finding. But, the medium is more appropriate because it doesn't necessarily lead to a loss of funds, the malicious user cannot increase the trust to infinity. It will always be equal to TRUST_AMOUNT which can be small, there's a maximum number of vouchers that one member can have.

Hence, planning to accept the escalation and validate with medium severity.

trachevgeorgi

> Then, I agree it has to be a valid finding. But, the medium is more appropriate because it doesn't necessarily lead to a loss of funds, the malicious user cannot increase the trust to infinity. It will always be equal to TRUST_AMOUNT which can be small, there's a maximum number of vouchers that one member can have.
>
> Hence, planning to accept the escalation and validate with medium severity.

I am ok with a medium evaluation. I will not be escalating further.

WangSecurity

Result: Medium Unique

@mystery0x @trachevgeorgi are there any duplicates?

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- trachevgeorgi: accepted

Issue M-4: Minimum borrow amount can be surpassed and borrower can be treated as being overdue earlier than their actual overdue time

Source: https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/114

Found by

Bigsam, hyh

Summary

It is possible to borrow less than `_minBorrow` and preliminary be marked as overdue when `assetManager` have temporary fund access limitations.

Vulnerability Detail

UToken's `borrow()` can be effectively run with lesser amount than `_minBorrow` when it is a liquidity shortage in the asset manager's underlying markets and they can return only some dust amount or nothing at all. In these cases `borrow()` call will still be concluded. Particularly, it is possible to run it with zero amount when `assetManager` cannot access liquidity.

In that case the borrower, if they borrow for the first time after full repay, will not have their `lastRepay` field reset on a subsequent material borrow operations as it will already be set on zero amount borrow before. As a result such borrowers can be effectively overdue for the system way before the actual overdue time passes for them.

Impact

`_minBorrow` threshold can be violated when market conditions restrict `assetManager` withdrawals. A user can have `lastRepay` set earlier than time of obtaining the funds, which will mark them overdue before the actual overdue time comes by. This will have a material adverse impact both on such a borrower (for them `checkIsOverdue` will be true, so they won't be able to borrow or create vouches) and their lenders (for them `stakerFrozen` and `frozenCoinAge` will be increased and staking rewards diminished).

Code Snippet

If current market conditions don't allow any material withdrawal then `borrow()` still can happen and `lastRepay` be set on any dust or even zero amount being lent out:

UToken.sol#L611-L634

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

        // Calculate the origination fee
        uint256 fee = calculatingFee(actualAmount);

        if (_borrowBalanceView(msg.sender) + actualAmount + fee > _maxBorrow)
↪   revert AmountExceedMaxBorrow();
        if (checkIsOverdue(msg.sender)) revert MemberIsOverdue();
        if (amount > assetManagerContract.getLoanableAmount(underlying)) revert
↪   InsufficientFundsLeft();
        if (!accrueInterest()) revert AccrueInterestFailed();

        uint256 borrowedAmount = borrowBalanceStoredInternal(msg.sender);

        // Initialize the last repayment date to the current block timestamp
>>      if (getLastRepay(msg.sender) == 0) {
            accountBorrows[msg.sender].lastRepay = getTimestamp();
        }

        // Withdraw the borrowed amount of tokens from the assetManager and send
↪   them to the borrower
>>      uint256 remaining = assetManagerContract.withdraw(underlying, to,
↪   amount);
>>      if (remaining > amount) revert WithdrawFailed();
>>      actualAmount -= decimalScaling(remaining, underlyingDecimal);
```

If market is such that `assetManagerContract.withdraw` can only withdraw dust or can't withdraw anything, a user can request to borrow an amount bigger than minimal, but `borrow()` will be executed with some dust or even zero amount effectively borrowed. This isn't fully covered by the `getLoanableAmount()` check since it measures total funds invested via `getSupplyView()` calls to the underlying markets.

As `_minBorrow` is for amount effectively borrowed, and not just for amount requested, it will be in a violation:

UToken.sol#L141-L144

```
    /**
>>   *  @dev Min amount that can be borrowed by a single member
     */
    uint256 private _minBorrow;
```

Also, it will have a side effect of resetting `lastRepay` even with zero amount borrowed when the borrower had no debt as of time of the call. This will effectively mark a borrower as an overdue when time since they obtained any material debt is in fact much less than `overdueTime`:

UToken.sol#L459-L465

```
    function checkIsOverdue(address account) public view override returns (bool
↪   isOverdue) {
        if (_getBorrowed(account) != 0) {
>>          uint256 lastRepay = getLastRepay(account);
>>          uint256 diff = getTimestamp() - lastRepay;
>>          isOverdue = overdueTime < diff;
        }
    }
```

UToken.sol#L450-L452

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

This can happen as subsequent `borrow()` calls will not set `lastRepay` as the logic is based on having empty `lastRepay`:

UToken.sol#L627-L629

```
if (getLastRepay(msg.sender) == 0) {
    accountBorrows[msg.sender].lastRepay = getTimestamp();
}
```

Tool used

Manual Review

Recommendation

Consider controlling the effective amount being borrowed, e.g.:

UToken.sol#L611-L634

```
    function borrow(address to, uint256 amount) external override
↪   onlyMember(msg.sender) whenNotPaused nonReentrant {
        IAssetManager assetManagerContract = IAssetManager(assetManager);
        uint256 actualAmount = decimalScaling(amount, underlyingDecimal);
-       if (actualAmount < _minBorrow) revert AmountLessMinBorrow();

        // Calculate the origination fee
        uint256 fee = calculatingFee(actualAmount);

        if (_borrowBalanceView(msg.sender) + actualAmount + fee > _maxBorrow)
↪   revert AmountExceedMaxBorrow();
        if (checkIsOverdue(msg.sender)) revert MemberIsOverdue();
        if (amount > assetManagerContract.getLoanableAmount(underlying)) revert
↪   InsufficientFundsLeft();
        if (!accrueInterest()) revert AccrueInterestFailed();

        uint256 borrowedAmount = borrowBalanceStoredInternal(msg.sender);

        // Initialize the last repayment date to the current block timestamp
        if (getLastRepay(msg.sender) == 0) {
            accountBorrows[msg.sender].lastRepay = getTimestamp();
        }

        // Withdraw the borrowed amount of tokens from the assetManager and send
↪   them to the borrower
        uint256 remaining = assetManagerContract.withdraw(underlying, to,
↪   amount);
        if (remaining > amount) revert WithdrawFailed();
        actualAmount -= decimalScaling(remaining, underlyingDecimal);
+       if (actualAmount < _minBorrow) revert AmountLessMinBorrow();
```

Discussion

maxweng

I think this issue only happens when the underlying market protocol we integrate doesn't implement the `withdraw()` function correctly, which it succeeded and returned true but the withdrawal amount was less than what the caller requested. But to tight things up on our end, I think we can add another check on the actual withdrawal amount.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/unioncredit/union-v2-contracts/pull/175

dmitriia

> The protocol team fixed this issue in the following PRs/commits:
> unioncredit/union-v2-contracts#175

Fix looks ok

Issue M-5: `updateLocked()` locks a rounded down value

Source: https://github.com/sherlock-audit/2024-06-union-finance-update-2-judging/issues/133

Found by

hyh

Summary

Since `decimalReducing()` rounds down and fee can use all the precision space the UToken's `updateLocked()` call performed on borrowing will effectively lock less then is borrowed.

Vulnerability Detail

`decimalReducing(actualAmount + fee, underlyingDecimal)` performed on locking can lose precision, i.e. since fee is added the rounding down can have a material impact.

Impact

User can have borrowed value slightly exceeding the cumulative lock amount due to rounding of the fee added.

Code Snippet

`updateLocked()` will lock a rounded down number for a user:

UToken.sol#L656-L660

```
IUserManager(userManager).updateLocked(
    msg.sender,
    decimalReducing(actualAmount + fee, underlyingDecimal),
    true
);
```

Tool used

Manual Review

Recommendation

Consider introducing an option for rounding the `decimalReducing()` output up, e.g.:

UToken.sol#L656-L660

```
        IUserManager(userManager).updateLocked(
            msg.sender,
-           decimalReducing(actualAmount + fee, underlyingDecimal),
+           decimalReducing(actualAmount + fee, underlyingDecimal, true),
            true
        );
```

https://github.com/sherlock-audit/2024-06-union-finance-update-2/blob/main/union-v2-contracts/contracts/ScaledDecimalBase.sol#L19-L27

```
-   function decimalReducing(uint256 actualAmount, uint8 decimal) internal pure
 ↪  returns (uint256) {
+   function decimalReducing(uint256 actualAmount, uint8 decimal, bool roundUp)
 ↪  internal pure returns (uint256) {
        if (decimal > 18) {
            uint8 diff = decimal - 18;
            return actualAmount * 10 ** diff;
        } else {
            uint8 diff = 18 - decimal;
            uint256 rounding = roundUp ? 10 ** diff - 1 : 0;
-           return actualAmount / 10 ** diff;
+           return (actualAmount + rounding) / 10 ** diff;
        }
    }
```

Discussion

sherlock-admin3

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

0xmystery commented:

> Low QA on rounding direction

dmitriia

> Low QA on rounding direction

Similarly, there is no such rule.

dmitriia

Escalate It looks like the `locked < principal` state reached with locking slightly less than it was borrowed leads to inability to fully close a position with write off as, since `repayAmount` is locked amount derived, it will be less than principal:

UserManager.sol#L866-L867

```
        IAssetManager(assetManager).debtWriteOff(stakingToken, amount);
>>      uToken.debtWriteOff(borrowerAddress, amount);
```

UToken.sol#L785-L800

```
    function debtWriteOff(address borrower, uint256 amount) external override
↪   whenNotPaused onlyUserManager {
        if (amount == 0) revert AmountZero();
        uint256 actualAmount = decimalScaling(amount, underlyingDecimal);

        uint256 oldPrincipal = _getBorrowed(borrower);
        uint256 repayAmount = actualAmount > oldPrincipal ? oldPrincipal :
↪   actualAmount;

        accountBorrows[borrower].principal = oldPrincipal - repayAmount;
        _totalBorrows -= repayAmount;

>>      if (repayAmount == oldPrincipal) {
            // If all principal is written off, we can reset the last repaid
↪   time to 0.
            // which indicates that the borrower has no outstanding loans.
            accountBorrows[borrower].lastRepay = 0;
        }
    }
```

Since it breaks this workflow and `lastRepay` has a material impact on the users (this borrower will have `checkIsOverdue() == true` for a next borrow, i.e. will not be able to create a non-dust position thereafter), it looks like it's a medium probability (writing-off is a requirement) and impact (credit line is unavailable with the corresponding material impact on the borrower), i.e. it's a medium severity issue.

sherlock-admin3

> Escalate It looks like the `locked < principal` state reached with locking slightly less than it was borrowed leads to inability to fully close a position with write off as, since `repayAmount` is locked amount derived, it will be less than principal:
>
> UserManager.sol#L866-L867
>
> ```
>         IAssetManager(assetManager).debtWriteOff(stakingToken, amount);
> ```

```
>>      uToken.debtWriteOff(borrowerAddress, amount);
```

UToken.sol#L785-L800

```
    function debtWriteOff(address borrower, uint256 amount) external
↪   override whenNotPaused onlyUserManager {
        if (amount == 0) revert AmountZero();
        uint256 actualAmount = decimalScaling(amount, underlyingDecimal);

        uint256 oldPrincipal = _getBorrowed(borrower);
        uint256 repayAmount = actualAmount > oldPrincipal ? oldPrincipal :
↪   actualAmount;

        accountBorrows[borrower].principal = oldPrincipal - repayAmount;
        _totalBorrows -= repayAmount;

>>      if (repayAmount == oldPrincipal) {
            // If all principal is written off, we can reset the last
↪   repaid time to 0.
            // which indicates that the borrower has no outstanding loans.
            accountBorrows[borrower].lastRepay = 0;
        }
    }
```

Since it breaks this workflow and `lastRepay` has a material impact on the users (this borrower will have `checkIsOverdue() == true` right after next borrow, i.e. for a non-dust position), then it has medium probability and impact, i.e. it's a medium severity issue.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

mystery0x

Please see comment on #122.

dmitriia

> Please see comment on #122.

It's not relevant here as, while the rounding is not limited to 1 wei, the issue is not about the size of rounding, but about the logic break up. The logic expects precise amounts match and is not working otherwise (`lastRepay` isn't set).

WangSecurity

This report is about precision loss and it falls under the following rule:

> PoC is required for all issues falling into any of the following groups:
> issues related to precision loss

I believe this report lacks POC (coded or written) and sufficient proof rounding down precision loss.

Planning to reject the escalation and leave the issue as it is, since it doesn't fulfil the POC requirements.

dmitriia

Schematic POC:

0. Let's examine USDC market with originationFee = `1e-8`, which is `1e10` in 18 dp representation. Bob has no loans, Alice opened trust for him only
1. Bob borrows `99` USDC from Alice's trust, his principal is set to this amount converted to 18 dp, `actualAmount`, with `calculatingFee(actualAmount) = (originationFee * actualAmount) / WAD` added on top. The principal for their loan position is `99e18 + 1e10 * 99e18 / 1e8 = 99 * (1e18 + 1e10)`
2. Locking operation deals with 6 dp figures and uses `decimalReducing(actualAmount + fee, underlyingDecimal)`, so it locks only `99 * (1e18 + 1e10) / 1e12 = 99e6`
3. A substantial time passes and this debt ends up being written off, for example via an arrangement between Bob and Alice for some OTC repayment, say Bob will give a similarly priced item to Alice off chain and she will write off the debt
4. Writing off uses `99e6` only (being based on underlying amount, 6 dp figure) and will not clear Bob's principal fully, he will still have the `99e10` part (it's an internal 18 dp figure). Since principal was not cleared, Bob's `lastRepay` was not reset, it still holds the value set in (1), i.e. the system treats his position as being borrowed long time ago and not repaid since. It's not material for now as position remainder is dust sized
5. However, when Bob try to borrow again from Alice's trust (which is tied to Bob's address) it will be denied as his position will be deemed overdue, so the existing trust will be unavailable for borrowing

WangSecurity

Unfortunately, POC and the proof of precision loss should have been described in the report. Hence, I cannot accept this as a rewarded finding based on the rules.

dmitriia

Well, in this case the issue can be excluded as incomplete per initial submission. Please refrain from implying the need of any additional actions in such cases in the

future. Note that excluding the issues from contest report also exclude them from the fix review, which scope is contest defined.

WangSecurity

> in this case the issue can be excluded as incomplete per initial submission

Good suggestion, I'm unsure it can be automated, but will look into that.

> Please refrain from implying the need of any additional actions in such cases in the future

Excuse me, I didn't imply anything is needed and it was poor phrasing from my side then.

> Note that excluding the issues from contest report also exclude them from the fix review, which scope is contest defined.

As far as I know, it doesn't and the sponsor can fix any issue they desire by just adding a "Will fix" label through their dashboard.

Hence, the decision remains the same, planning to reject the escalation and leave the issue as it is, based on the same arguments above.

dmitriia

Your can imply basically anything, there is no such obligation and I will be checking only issues from the report. If any others are deemed not important enough to be in the report, then their fix is not important either.

WangSecurity

After further considering the rule about the POC, decision is that submitting it after the contest when the lead judge requests or during the contest is sufficient. Excuse me for the confusion previously. The POC shows how the precision leads Bob's position being counted as opened and leaving the trust unavailable for Bob.

Planning to accept the escalation and validate with medium severity. @mystery0x @dmitriia are there any duplicates?

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- dmitriia: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/unioncredit/union-v2-contracts/pull/176

dmitriia

> The protocol team fixed this issue in the following PRs/commits:
> unioncredit/union-v2-contracts#176

Fix looks ok.

It looks like `1 wei` of locked will remain in some cases, as one described above, after `{borrow} -> {full repay}`, since borrow rounds up, while repay rounds down in `updateLocked(...)` arguments. As of now I see no issue in that as there won't be any material accumulation, this and further workflow remain operational, particularly `lastRepay` is correctly reset, being conditional on principal, which is fully cleared. Writing off will be functional as well due to `repayAmount = actualAmount > oldPrincipal ? oldPrincipal : actualAmount` logic.

Disclaimers

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

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

Note: Sherlock was made aware that ERC1155Voucher.sol and VouchFaucet.sol were accidentally included in the scope of this audit. These two contracts accounted for 1 High vulnerabilities and 3 Medium vulnerabilities in this audit report.