SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Blueberry
Prepared by: Sherlock
Lead Security Expert: obront
Dates Audited: February 6 - February 20, 2023
Prepared on: March 19, 2023

Introduction

Blueberry unifies the DeFi experience: Aggregating, Automating, and Boosting Capital Efficiency for top DeFi Strategies.

Scope

Github

```
Includes:
    - All contracts in `/contracts`
```

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
26	14

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

obront
0x52
0Kage
Robert
rvierdiiev
berndartmueller
koxuan
Jeiwan
mert_eren
carrot
cducrest-brainbot
rbserver
cergyk
chaduke
Ch_301
clems4ever
tives
y1cunhui

evan
Ruhum
GimelSec
peanuts
csanuragjain
minhtrng
ctf_sec
sakshamguruji
sinarette
banditx0x
psy4n0n
stent
tsvetanovv
Breeje
8olidity
Bauer
SPYBOY
PRAISE
Saeedalipoor01988
shark
0xChinedu
RaymondFam
XKET
saian
Dug
Avci
Nyx
WatchDogs
HonorLt
Aymen0909
Chinmay
hl_

Issue H-1: Too few ICHI v2 farming reward tokens transferred to the user due to incorrect decimal precision

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/319

Found by

0x52, berndartmueller

Summary

The `burn` function in the `WIchiFarm` contract transfers too few `ICHI` **v2** farming reward tokens to the caller due to using 9 decimals instead of 18 decimals for the `ICHI` **v2** token.

Vulnerability Detail

Closing an ICHI vault spell farming position burns the wrapped ICHI vault LP tokens (`WIchiFarm` ERC-1155 tokens). Farming rewards are harvested from the ICHI farm (see contract on Etherscan) and received as `ICHI` **v1** tokens.

The `ICHI` **v1** ERC-20 token uses **9 decimals** (see token on Etherscan), whereas the `ICHI` **v2** ERC-20 token uses **18 decimals** (see token on Etherscan).

Those received `ICHI` **v1** tokens are then converted to **v2** tokens in line 134.

To calculate the user's share of eligible `ICHI` **v2** reward tokens, the reward per share accumulator `stIchiPerShare` at the time of minting the `WIchiFarm` token and the current `enIchiPerShare` accumulator is used.

However, those accumulator values are in **9 decimals** precision (please see the `ichiFarmV2.harvest` function for proof that `pool.accIchiPerShare` uses 9 decimals, otherwise the `ICHI` token transfer would fail due to inflated `_pendingIchi`). Given that `amount` is in **18 decimals**, the calculation of `stIchi` and `enIchi` in lines 143 and 144 will result in a value with **9 decimals** precision.

As previously mentioned, the `ICHI` **v2** token uses **18 decimals**. Therefore, too few `ICHI` **v2** tokens are transferred.

Impact

Users will receive substantially fewer `ICHI` v2 farming reward tokens than expected.

Code Snippet

wrapper/WIchiFarm.sol#L143-L144

```
116: function burn(uint256 id, uint256 amount)
117:     external
118:     nonReentrant
119:     returns (uint256)
120: {
121:     if (amount == type(uint256).max) {
122:         amount = balanceOf(msg.sender, id);
123:     }
124:     (uint256 pid, uint256 stIchiPerShare) = decodeId(id);
125:     _burn(msg.sender, id, amount);
126:
127:     uint256 ichiRewards = ichiFarm.pendingIchi(pid, address(this));
128:     ichiFarm.harvest(pid, address(this));
129:     ichiFarm.withdraw(pid, amount, address(this));
130:
131:     // Convert Legacy ICHI to ICHI v2
132:     if (ichiRewards > 0) {
133:         ICHIv1.safeApprove(address(ICHI), ichiRewards);
134:         ICHI.convertToV2(ichiRewards);
135:     }
136:
137:     // Transfer LP Tokens
138:     address lpToken = ichiFarm.lpToken(pid);
139:     IERC20Upgradeable(lpToken).safeTransfer(msg.sender, amount);
140:
141:     // Transfer Reward Tokens
142:     (uint256 enIchiPerShare, , ) = ichiFarm.poolInfo(pid);
143:     uint256 stIchi = (stIchiPerShare * amount).divCeil(1e18);
144:     uint256 enIchi = (enIchiPerShare * amount) / 1e18; // @audit-info
↪  `enIchi` and `stIchi` are in 9 decimal precision
145:
146:     if (enIchi > stIchi) {
147:         ICHI.safeTransfer(msg.sender, enIchi - stIchi);
148:     }
149:     return pid;
150: }
```

Tool used

Manual Review

Recommendation

Consider changing the denominator in lines 143 and 144 from `1e18` to `1e9` to use the required `18` decimals for the `ICHI` v2 token.

Issue H-2: Deposit Theft by Crashing LP Spot Prices Through MEV

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/220

Found by

Robert

Summary

When depositing into an Ichi vault it allows a user to deposit all in a single token and determines the amount of vault shares they receive based off the price of that token in the second. This does not use twap but rather a combination of spot and twap in which it chooses the lesser of the two. There's protection against heavy manipulation occurring all on one block by checking if the difference between the two is greater than 5%, and failing if it is and if the last price change happened on the current block, but if the last price change was on a previous block it does not revert.

Multi-block MEV allows malicious actors to manipulate price over multiple blocks with no risk at all. They can easily manipulate token price down to near 0 on one block, a user tries to deposit on the next and gets almost $0 worth of vault shares for their tokens, vault shareholders pocket the extra tokens from the user's deposit, and token price is returned. With this, a user depositing into Blueberry could have their entire deposit stolen by a malicious actor.

This is fairly easy to do now if you see your own block coming up, manipulate price through MEV on the block before that, then include victim transaction and repayment on your own block right after that (not technically needing MMEV). It will be even easier in the future when MMEV is included in Flashbots.

Vulnerability Detail

0. Malicious attacker has validator or uses MMEV through flashbots.
1. Directly before a block they fully control, the validator manipulates token0 price in the LP pool to next to nothing.
2. On the next block, attacker flash loans a large amount of tokens to purchase Ichi Vault shares.
3. Attacker includes all pending user deposits into the pool that use that token. Each of these returns to the user almost nothing.
4. Attacker withdraws shares and included are the tokens that were stolen from users.

Impact

User deposits will be stolen.

Code Snippet

Price check only reverting on a large change if the block is the same as now: https://etherscan.io/token/0x2a8E09552782563f7A076ccec0Ff39473B91Cd8F#code#L2807

Amount of shares relying on price: https://etherscan.io/token/0x2a8E09552782563f7A076ccec0Ff39473B91Cd8F#code#L2829

Tool used

Manual Review

Recommendation

Check spot and twap price the same way IchiVault does but ensure they are within an allowed delta regardless of when price was last updated.

Issue H-3: Users who deposit extra funds into their Ichi farming positions will lose all their ICHI rewards

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/158

Found by

obront, sinarette, 0x52, tives, carrot, minhtrng, rvierdiiev, berndartmueller

Summary

When a user deposits extra funds into their Ichi farming position using `openPositionFarm()`, the old farming position will be closed down and a new one will be opened. Part of this process is that their ICHI rewards will be sent to the `IchiVaultSpell.sol` contract, but they will not be distributed. They will sit in the contract until the next user (or MEV bot) calls `closePositionFarm()`, at which point they will be stolen by that user.

Vulnerability Detail

When Ichi farming positions are opened via the `IchiVaultSpell.sol` contract, `openPositionFarm()` is called. It goes through the usual deposit function, but rather than staking the LP tokens directly, it calls `wIchiFarm.mint()`. This function deposits the token into the `ichiFarm`, encodes the deposit as an ERC1155, and sends that token back to the Spell:

```
function mint(uint256 pid, uint256 amount)
    external
    nonReentrant
    returns (uint256)
{
    address lpToken = ichiFarm.lpToken(pid);
    IERC20Upgradeable(lpToken).safeTransferFrom(
        msg.sender,
        address(this),
        amount
    );
    if (
        IERC20Upgradeable(lpToken).allowance(
            address(this),
            address(ichiFarm)
        ) != type(uint256).max
    ) {
        // We only need to do this once per pool, as LP token's allowance won't
↪   decrease if it's -1.
        IERC20Upgradeable(lpToken).safeApprove(
```

```
            address(ichiFarm),
            type(uint256).max
        );
    }
    ichiFarm.deposit(pid, amount, address(this));
    // @ok if accIchiPerShare is always changing, so how does this work?
    // it's basically just saving the accIchiPerShare at staking time, so when
↪   you unstake, it can calculate the difference
    // really fucking smart actually
    (uint256 ichiPerShare, , ) = ichiFarm.poolInfo(pid);
    uint256 id = encodeId(pid, ichiPerShare);
    _mint(msg.sender, id, amount, "");
    return id;
}
```

The resulting ERC1155 is posted as collateral in the Blueberry Bank.

If the user decides to add more funds to this position, they simply call `openPositionFarm()` again. The function has logic to check if there is already existing collateral of this LP token in the Blueberry Bank. If there is, it removes the collateral and calls `wIchiFarm.burn()` (which harvests the Ichi rewards and withdraws the LP tokens) before repeating the deposit process.

```
function burn(uint256 id, uint256 amount)
    external
    nonReentrant
    returns (uint256)
{
    if (amount == type(uint256).max) {
        amount = balanceOf(msg.sender, id);
    }
    (uint256 pid, uint256 stIchiPerShare) = decodeId(id);
    _burn(msg.sender, id, amount);

    uint256 ichiRewards = ichiFarm.pendingIchi(pid, address(this));
    ichiFarm.harvest(pid, address(this));
    ichiFarm.withdraw(pid, amount, address(this));

    // Convert Legacy ICHI to ICHI v2
    if (ichiRewards > 0) {
        ICHIv1.safeApprove(address(ICHI), ichiRewards);
        ICHI.convertToV2(ichiRewards);
    }

    // Transfer LP Tokens
    address lpToken = ichiFarm.lpToken(pid);
    IERC20Upgradeable(lpToken).safeTransfer(msg.sender, amount);
```

```
    // Transfer Reward Tokens
    (uint256 enIchiPerShare, , ) = ichiFarm.poolInfo(pid);
    uint256 stIchi = (stIchiPerShare * amount).divCeil(1e18);
    uint256 enIchi = (enIchiPerShare * amount) / 1e18;

    if (enIchi > stIchi) {
        ICHI.safeTransfer(msg.sender, enIchi - stIchi);
    }
    return pid;
}
```

However, this deposit process has no logic for distributing the ICHI rewards. Therefore, these rewards will remain sitting in the `IchiVaultSpell.sol` contract and will not reach the user.

For an example of how this is handled properly, we can look at the opposite function, `closePositionFarm()`. In this case, the same `wIchiFarm.burn()` function is called. But in this case, it's followed up with an explicit call to withdraw the ICHI from the contract to the user.

```
doRefund(ICHI);
```

This `doRefund()` function refunds the contract's full balance of ICHI to the `msg.sender`, so the result is that the next user to call `closePositionFarm()` will steal the ICHI tokens from the original user who added to their farming position.

Impact

Users who farm their Ichi LP tokens for ICHI rewards can permanently lose their rewards.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/IchiVaultSpell.sol#L199-L249

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/wrapper/WIchiFarm.sol#L116-L150

Here is a link to the `harvest()` function on the IchiFarmV2.sol contract, which is called by `wIchiFarm.sol` and contains the logic for distributing ICHI rewards: https://github.com/ichifarm/ichi-farming/blob/206c44b790fbb2a1e3a655685eb3ab8d793c9f00/contracts/ichiFarmV2.sol#L238-L257

Tool used

Manual Review

Recommendation

In the `openPositionFarm()` function, in the section that deals with withdrawing existing collateral, add a line that claims the ICHI rewards for the calling user.

```
if (collSize > 0) {
    (uint256 decodedPid, ) = wIchiFarm.decodeId(collId);
    if (farmingPid != decodedPid) revert INCORRECT_PID(farmingPid);
    if (posCollToken != address(wIchiFarm))
        revert INCORRECT_COLTOKEN(posCollToken);
    bank.takeCollateral(collSize);
    wIchiFarm.burn(collId, collSize);
+   doRefund(ICHI);
}
```

Issue H-4: LP tokens cannot be valued because ICHI cannot be priced by oracle, causing all new open positions to revert

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/152

Found by

obront

Summary

In order to value ICHI LP tokens, the oracle uses the Fair LP Pricing technique, which uses the prices of both individual tokens, along with the quantities, to calculate the LP token value. However, this process requires the underlying token prices to be accessible by the oracle. Both Chainlink and Band do not support the ICHI token, so the function will fail, causing all new positions using the IchiVaultSpell to revert.

Vulnerability Detail

When a new Ichi position is opened, the ICHI LP tokens are posted as collateral. Their value is assessed using the `IchiLpOracle#getPrice()` function:

```
function getPrice(address token) external view override returns (uint256) {
    IICHIVault vault = IICHIVault(token);
    uint256 totalSupply = vault.totalSupply();
    if (totalSupply == 0) return 0;

    address token0 = vault.token0();
    address token1 = vault.token1();

    (uint256 r0, uint256 r1) = vault.getTotalAmounts();
    uint256 px0 = base.getPrice(address(token0));
    uint256 px1 = base.getPrice(address(token1));
    uint256 t0Decimal = IERC20Metadata(token0).decimals();
    uint256 t1Decimal = IERC20Metadata(token1).decimals();

    uint256 totalReserve = (r0 * px0) /
        10**t0Decimal +
        (r1 * px1) /
        10**t1Decimal;

    return (totalReserve * 1e18) / totalSupply;
}
```

This function uses the "Fair LP Pricing" formula, made famous by Alpha Homora. To simplify, this uses an oracle to get the prices of both underlying tokens, and then calculates the LP price based on these values and the reserves.

However, this process requires that we have a functioning oracle for the underlying tokens. However, Chainlink and Band both do not support the ICHI token (see the links for their comprehensive lists of data feeds). As a result, the call to `base.getPrice(token0)` will fail.

All prices are calculated in the `isLiquidatable()` check at the end of the `execute()` function. As a result, any attempt to open a new ICHI position and post the LP tokens as collateral (which happens in both `openPosition()` and `openPositionFarm()`) will revert.

Impact

All new positions opened using the `IchiVaultSpell` will revert when they attempt to look up the LP token price, rendering the protocol useless.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/oracle/IchiLpOracle.sol#L19-L39

Tool used

Manual Review

Recommendation

There will need to be an alternate form of oracle that can price the ICHI token. The best way to accomplish this is likely to use a TWAP of the price on an AMM.

Discussion

Gornutz

There is additional oracles for assets not supported by chainlink or band but just not in scope of this specific audit.

hrishibhat

Based on the context there are no implementations for getting the price of the ICHI token. Considering this a valid issue.

Issue H-5: LP tokens are not sent back to withdrawing user

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/151

Found by

Dug, chaduke, obront, sinarette, evan, 0x52, Bauer, koxuan, minhtrng, cergyk, Jeiwan, rvierdiiev, Ch_301, berndartmueller

Summary

When users withdraw their assets from `IchiVaultSpell.sol`, the function unwinds their position and sends them back their assets, but it never sends them back the amount they requested to withdraw, leaving the tokens stuck in the Spell contract.

Vulnerability Detail

When a user withdraws from `IchiVaultSpell.sol`, they either call `closePosition()` or `closePositionFarm()`, both of which make an internal call to `withdrawInternal()`.

The following arguments are passed to the function:

- strategyId: an index into the `strategies` array, which specifies the Ichi vault in question
- collToken: the underlying token, which is withdrawn from Compound
- amountShareWithdraw: the number of underlying tokens to withdraw from Compound
- borrowToken: the token that was borrowed from Compound to create the position, one of the underlying tokens of the vault
- amountRepay: the amount of the borrow token to repay to Compound
- amountLpWithdraw: the amount of the LP token to withdraw, rather than trade back into borrow tokens

In order to accomplish these goals, the contract does the following...

1) Removes the LP tokens from the ERC1155 holding them for collateral.

```
doTakeCollateral(strategies[strategyId].vault, lpTakeAmt);
```

2) Calculates the number of LP tokens to withdraw from the vault.

```
uint256 amtLPToRemove = vault.balanceOf(address(this)) - amountLpWithdraw;
vault.withdraw(amtLPToRemove, address(this));
```

3) Converts the non-borrowed token that was withdrawn in the borrowed token (not copying the code in, as it's not relevant to this issue).
4) Withdraw the underlying token from Compound.

```
doWithdraw(collToken, amountShareWithdraw);
```

5) Pay back the borrowed token to Compound.

```
doRepay(borrowToken, amountRepay);
```

6) Validate that this situation does not put us above the maxLTV for our loans.

```
_validateMaxLTV(strategyId);
```

7) Sends the remaining borrow token that weren't paid back and withdrawn underlying tokens to the user.

```
doRefund(borrowToken);
doRefund(collToken);
```

Crucially, the step of sending the remaining LP tokens to the user is skipped, even though the function specifically does the calculations to ensure that `amountLpWithdraw` is held back from being taken out of the vault.

Impact

Users who close their positions and choose to keep LP tokens (rather than unwinding the position for the constituent tokens) will have their LP tokens stuck permanently in the IchiVaultSpell contract.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/IchiVaultSpell.sol#L276-L330

Tool used

Manual Review

Recommendation

Add an additional line to the `withdrawInternal()` function to refund all LP tokens as well:

```
  doRefund(borrowToken);
  doRefund(collToken);
+ doRefund(address(vault));
```

Discussion

Gornutz

duplicate of 34

Issue H-6: Fail to accrue interests on multiple token positions

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/140

Found by

Jeiwan, cducrest-brainbot, rvierdiiev

Summary

In `BlueBerryBank.sol` the functions `borrow`, `repay`, `lend`, or `withdrawLend` call `poke(token)` to trigger interest accrual on concerned token, but fail to do so for other token debts of the concerned position. This could lead to wrong calculation of position's debt and whether the position is liquidatable.

Vulnerability Detail

Whether a position is liquidatable or not is checked at the end of the `execute` function, the execution should revert if the position is liquidatable.

The calculation of whether a position is liquidatable takes into account all the different debt tokens within the position. However, the debt accrual has been triggered only for one of these tokens, the one concerned by the executed action. For other tokens, the value of `bank.totalDebt` will be lower than what it should be. This results in the debt value of the position being lower than what it should be and a position seen as not liquidatable while it should be liquidatable.

Impact

Users may be able to operate on their position leading them in a virtually liquidatable state while not reverting as interests were not applied. This will worsen the debt situation of the bank and lead to overall more liquidatable positions.

Code Snippet

execute checking isLiquidatable without triggering interests:

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L607

actions only poke one token (here for borrow):

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L709-L715

bank.totalDebt is used to calculate a position's debt while looping over every tokens:

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L451-L475

The position's debt is used to calculate the risk:

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L477-L495

The risk is used to calculate whether a debt is liquidatable:

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L497-L505

Tool used

Manual Review

Recommendation

Review how token interests are triggered. Probably need to accrue interests on every debt token of a position at the beginning of execute.

Issue H-7: Users can get around MaxLTV because of lack of strategyId validation

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/129

Found by

obront, carrot, cergyk, Jeiwan, Ch_301

Summary

When a user withdraws some of their underlying token, there is a check to ensure they still meet the Max LTV requirements. However, they are able to arbitrarily enter any `strategyId` that they would like for this check, which could allow them to exceed the LTV for their real strategy while passing the approval.

Vulnerability Detail

When a user calls `IchiVaultSpell.sol#reducePosition()`, it removes some of their underlying token from the vault, increasing the LTV of any loans they have taken.

As a result, the `_validateMaxLTV(strategyId)` function is called to ensure they remain compliant with their strategy's specified LTV:

```
function _validateMaxLTV(uint256 strategyId) internal view {
    uint256 debtValue = bank.getDebtValue(bank.POSITION_ID());
    (, address collToken, uint256 collAmount, , , , , ) = bank
        .getCurrentPositionInfo();
    uint256 collPrice = bank.oracle().getPrice(collToken);
    uint256 collValue = (collPrice * collAmount) /
        10**IERC20Metadata(collToken).decimals();

    if (
        debtValue >
        (collValue * maxLTV[strategyId][collToken]) / DENOMINATOR
    ) revert EXCEED_MAX_LTV();
}
```

To summarize, this check:

- Pulls the position's total debt value
- Pulls the position's total value of underlying tokens
- Pulls the specified maxLTV for this strategyId and underlying token combination

- Ensures that `underlyingTokenValue * maxLTV > debtValue`

But there is no check to ensure that this `strategyId` value corresponds to the strategy the user is actually invested in, as we can see the `reducePosition()` function:

```
function reducePosition(
    uint256 strategyId,
    address collToken,
    uint256 collAmount
) external {
    doWithdraw(collToken, collAmount);
    doRefund(collToken);
    _validateMaxLTV(strategyId);
}
```

Here is a quick proof of concept to explain the risk:

- Let's say a user deposits 1000 DAI as their underlying collateral.
- They are using a risky strategy (let's call it strategy 911) which requires a maxLTV of 2X (ie `maxLTV[911][DAI] = 2e5`)
- There is another safer strategy (let's call it strategy 411) which has a maxLTV of 5X (ie `maxLTV[411][DAI] = 4e5`)
- The user takes the max loan from the risky strategy, borrowing $2000 USD of value.
- They are not allowed to take any more loans from that strategy, or remove any of their collateral.
- Then, they call `reducePosition()`, withdrawing 1600 DAI and entering `411` as the strategyId.
- The `_validateMaxLTV` check will happen on `strategyId = 411`, and will pass, but the result will be that the user now has only 400 DAI of underlying collateral protecting $2000 USD worth of the risky strategy, violating the LTV.

Impact

Users can get around the specific LTVs and create significantly higher leverage bets than the protocol has allowed. This could cause the protocol to get underwater, as the high leverage combined with risky assets could lead to dramatic price swings without adequate time for the liquidation mechanism to successfully protect solvency.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/IchiVaultSpell.sol#L266-L274

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/IchiVaultSpell.sol#L101-L113

Tool used

Manual Review

Recommendation

Since the collateral a position holds will always be the vault token of the strategy they have used, you can validate the `strategyId` against the user's collateral, as follows:

```
address positionCollToken = bank.positions(bank.POSITION_ID()).collToken;
address positionCollId = bank.positions(bank.POSITION_ID()).collId;
address unwrappedCollToken =
↪   IERC20Wrapper(positionCollToken).getUnderlyingToken(positionCollId);
require(strategies[strategyId].vault == unwrappedCollToken, "wrong strategy");
```

Issue H-8: Liquidator can take all collateral and underlying tokens for a fraction of the correct price

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/127

Found by

0x52, berndartmueller, obront

Summary

When performing liquidation calculations, we use the proportion of the individual token's debt they pay off to calculate the proportion of the liquidated user's collateral and underlying tokens to send to them. In the event that the user has multiple types of debt, the liquidator will be dramatically overpaid.

Vulnerability Detail

When a position's risk rating falls below the underlying token's liquidation threshold, the position becomes liquidatable. At this point, anyone can call `liquidate()` and pay back a share of their debt, and receive a propotionate share of their underlying assets.

This is calculated as follows:

```
uint256 oldShare = pos.debtShareOf[debtToken];
(uint256 amountPaid, uint256 share) = repayInternal(
    positionId,
    debtToken,
    amountCall
);

uint256 liqSize = (pos.collateralSize * share) / oldShare;
uint256 uTokenSize = (pos.underlyingAmount * share) / oldShare;
uint256 uVaultShare = (pos.underlyingVaultShare * share) / oldShare;

pos.collateralSize -= liqSize;
pos.underlyingAmount -= uTokenSize;
pos.underlyingVaultShare -= uVaultShare;

// ...transfer liqSize wrapped LP Tokens and uVaultShare underlying vault shares
↪  to the liquidator
}
```

To summarize:

- The liquidator inputs a debtToken to pay off and an amount to pay
- We check the amount of debt shares the position has on that debtToken
- We call `repayInternal()`, which pays off the position and returns the amount paid and number of shares paid off
- We then calculate the proportion of collateral and underlying tokens to give the liquidator
- We adjust the liquidated position's balances, and send the funds to the liquidator

The problem comes in the calculations. The amount paid to the liquidator is calculated as:

```
uint256 liqSize = (pos.collateralSize * share) / oldShare
uint256 uTokenSize = (pos.underlyingAmount * share) / oldShare;
uint256 uVaultShare = (pos.underlyingVaultShare * share) / oldShare;
```

These calculations are taking the total size of the collateral or underlying token. They are then multiplying it by `share / oldShare`. But `share / oldShare` is just the proportion of that one type of debt that was paid off, not of the user's entire debt pool.

Let's walk through a specific scenario of how this might be exploited:

- User deposits 1mm DAI (underlying) and uses it to borrow $950k of ETH and $50k worth of ICHI (11.8k ICHI)
- Both assets are deposited into the ETH-ICHI pool, yielding the same collateral token
- Both prices crash down by 25% so the position is now liquidatable (worth $750k)
- A liquidator pays back the full ICHI position, and the calculations above yield `pos.collateralSize * 11.8k / 11.8k` (same calculation for the other two formulas)
- The result is that for 11.8k ICHI (worth $37.5k after the price crash), the liquidator got all the DAI (value $1mm) and LP tokens (value $750k)

Impact

If a position with multiple borrows goes into liquidation, the liquidator can pay off the smallest token (guaranteed to be less than half the total value) to take the full position, stealing funds from innocent users.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L511-L572

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L760-L787

Tool used

Manual Review

Recommendation

Adjust these calculations to use `amountPaid / getDebtValue(positionId)`, which is accurately calculate the proportion of the total debt paid off.

Issue H-9: Users can be liquidated prematurely because calculation understates value of underlying position

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/126

Found by

obront

Summary

When the value of the underlying asset is calculated in `getPositionRisk()`, it uses the `underlyingAmount`, which is the amount of tokens initially deposited, without any adjustment for the interest earned. This can result in users being liquidated early, because the system undervalues their assets.

Vulnerability Detail

A position is considered liquidatable if it meets the following criteria:

```
((borrowsValue - collateralValue) / underlyingValue) >= underlyingLiqThreshold
```

The value of the underlying tokens is a major factor in this calculation. However, the calculation of the underlying value is performed with the following function call:

```
uint256 cv = oracle.getUnderlyingValue(
    pos.underlyingToken,
    pos.underlyingAmount
);
```

If we trace it back, we can see that `pos.underlyingAmount` is set when `lend()` is called (ie when underlying assets are deposited). This is the only place in the code where this value is moved upward, and it is only increased by the amount deposited. It is never moved up to account for the interest payments made on the deposit, which can materially change the value.

Impact

Users can be liquidated prematurely because the value of their underlying assets are calculated incorrectly.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L485-L488

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/oracle/CoreOracle.sol#L182-L189

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L644

Tool used

Manual Review

Recommendation

Value of the underlying assets should be derived from the vault shares and value, rather than being stored directly.

Issue H-10: Interest component of underlying amount is not withdrawable using the `withdrawLend` function. Such amount is permanently locked in the BlueBerryBank contract

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/109

Found by

XKET, saian, chaduke, Ruhum, 0Kage, stent, GimelSec, carrot, minhtrng, koxuan, cergyk, Jeiwan, rbserver, berndartmueller

Summary

Soft vault shares are issued against interest bearing tokens issued by `Compound` protocol in exchange for underlying deposits. However, `withdrawLend` function caps the withdrawable amount to initial underlying deposited by user (`pos.underlyingAmount`). Capping underlying amount to initial underlying deposited would mean that a user can burn all his vault shares in `withdrawLend` function and only receive original underlying deposited.

Interest accrued component received from Soft vault (that rightfully belongs to the user) is no longer retrievable because the underlying vault shares are already burnt. Loss to the users is permanent as such interest amount sits permanently locked in Blueberry bank.

Vulnerability Detail

withdrawLend function in BlueBerryBank allows users to withdraw underlying amount from `Hard` or `Soft` vaults. `Soft` vault shares are backed by interest bearing `cTokens` issued by Compound Protocol

User can request underlying by specifying `shareAmount`. When user tries to send the maximum `shareAmount` to withdraw all the lent amount, notice that the amount withdrawable is limited to the `pos.underlyingAmount` (original deposit made by the user).

While this is the case, notice also that the full `shareAmount` is deducted from `underlyingVaultShare`. User cannot recover remaining funds because in the next call, user doesn't have any vault shares against his address. Interest accrued component on the underlying that was returned by `SoftVault` to `BlueberryBank` never makes it back to the original lender.

```
wAmount = wAmount > pos.underlyingAmount
        ? pos.underlyingAmount
```

```
        : wAmount;

    pos.underlyingVaultShare -= shareAmount;
    pos.underlyingAmount -= wAmount;
    bank.totalLend -= wAmount;
```

Impact

Every time, user withdraws underlying from a Soft vault, interest component gets trapped in BlueBerry contract. Here is a scenario.

- Alice deposits 1000 USDC into `SoftVault` using the `lend` function of BlueberryBank at T=0
- USDC soft vault mints 1000 shares to Blueberry bank
- USDC soft vault deposits 1000 USDC into Compound & receives 1000 cUSDC
- Alice at T=60 days requests withdrawal against 1000 Soft vault shares
- Soft Vault burns 1000 soft vault shares and requests withdrawal from Compound against 1000 cTokens
- Soft vault receives 1050 USDC (50 USDC interest) and sends this to BlueberryBank
- Blueberry Bank caps the withdrawal amount to 1000 (original deposit)
- Blueberry Bank deducts 0.5% withdrawal fees and deposits 995 USDC back to user

In the whole process, Alice has lost access to 50 USDC.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L693

Tool used

Manual Review

Recommendation

Introduced a new variable to adjust positions & removed cap on withdraw amount.

Highlighted changes I recommend to withdrawLend with //******//.

```
function withdrawLend(address token, uint256 shareAmount)
        external
```

```
        override
        inExec
        poke(token)
    {
        Position storage pos = positions[POSITION_ID];
        Bank storage bank = banks[token];
        if (token != pos.underlyingToken) revert INVALID_UTOKEN(token);

        //*********-audit cap shareAmount to maximum value,
↪   pos.underlyingVaultShare*******
        if (shareAmount > pos.underlyingVaultShare) {
            shareAmount = pos.underlyingVaultShare;
        }

        // if (shareAmount == type(uint256).max) {
        //     shareAmount = pos.underlyingVaultShare;
        // }

        uint256 wAmount;
        uint256 amountToOffset; //*********- audit added this to adjust
↪   position********
        if (address(ISoftVault(bank.softVault).uToken()) == token) {
            ISoftVault(bank.softVault).approve(
                bank.softVault,
                type(uint256).max
            );
            wAmount = ISoftVault(bank.softVault).withdraw(shareAmount);
        } else {
            wAmount = IHardVault(bank.hardVault).withdraw(token, shareAmount);
        }

        //*********- audit calculate amountToOffset********
        //*********-audit not capping wAmount anymore*******
        amountToOffset = wAmount > pos.underlyingAmount
            ? pos.underlyingAmount
            : wAmount;

        pos.underlyingVaultShare -= shareAmount;
     //*********-audit subtract amountToOffset instead of wAmount*******
        pos.underlyingAmount -= amountToOffset;
        bank.totalLend -= amountToOffset;

        wAmount = doCutWithdrawFee(token, wAmount);

        IERC20Upgradeable(token).safeTransfer(msg.sender, wAmount);
    }
```

Issue H-11: `isLiquidatable` function underprices risk, potentially preventing (or delaying) liquidations of under-collateralized positions

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/102

Found by

0Kage

Summary

Blueberry bank contract has a `isLiquidatable` function that checks if a given position is undercollateralized, and needs to be liquidated. Liquidators will call this function to check if the debt + strategy losses for a given position as a proportion of posted collateral exceed the liquidation threshold.

This function computes the `risk` of a position using its debt value, collateral value and underlying value. Debt value uses the unaccrued total debt that ignores the outstanding interest accrued on `cToken` borrowings. As a result, debt value used for risk calculation is lesser than actual value. This leads to an `underpricing` of risk that can prevent or delay liquidators from initiating a liquidation action leading to under-collateralization & protocol losses.

Vulnerability Detail

isLiquidatable function calculates risk of a given position by computing debt value, position value and underlying value in the `getPositionRisk` function.

However, debt value calculated by getDebtValue function does not include the interest accrued on `cTokens` on the outstanding debt.

```
function getDebtValue(uint256 positionId)
    public
    view
    override
    returns (uint256)
{
    uint256 value = 0;
    Position storage pos = positions[positionId];
    uint256 bitMap = pos.debtMap;
    uint256 idx = 0;
    while (bitMap > 0) {
        if ((bitMap & 1) != 0) {
            address token = allBanks[idx];
            uint256 share = pos.debtShareOf[token];
```

```
            Bank storage bank = banks[token];
            uint256 debt = (share * bank.totalDebt).divCeil(bank.totalShare);
↪   //-audit totalDebt here does not include the accrued component since last
↪   update
            value += oracle.getDebtValue(token, debt);
        }
        idx++;
        bitMap >>= 1;
    }
    return value;
}
```

As a result, debt value used for `risk` calculations is always lesser than or equal to actual debt outstanding. A lower risk value can generate a false signal to liquidators that a position is sufficiently collaterized. In reality, such a loan is under collateralized and should have gone for liquidation. (Note that `isLiquidatable` called within `Liquidate` function is correct value, because the function calls `poke` modifier that updates debt value)

Impact

Incases where significant time has elapsed since last debt update, `accrued` component could be significant in comparison to total debt. While unaccrued debt value might generate a `risk` value below liquidation threshold, including accrued component might exceed that threshold. This edge case is dangerous because it gives a false comfort to liquidators that position is sufficiently collateralized.

POC:

- Say Bank has total debt of 1000 USDC
- Alice borrows 80 USDC (8%) by posting 100 USDC worth ETH as collateral
- After 30 days, say total accrued is 50 USDC
- Current Risk calculation uses Alice debt value as 80 USDC
- Actual debt value of Alice is 8% * (1000 + 50) = 84 USDC
- All other things being equal (position losses + collateral posted), protocol underpriced risk of Alice positions

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L503

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L466

Tool used

Manual Review

Recommendation

Recommend replacing line 466 below

```
uint256 debt = (share * bank.totalDebt).divCeil(bank.totalShare)
```

with following

```
uint256 debt = (share * ICErc20(bank.cToken).borrowBalanceCurrent(address(this))
↪  ).divCeil(bank.totalShare)
```

This uses the latest debt value at the current timestamp for calculating risk of the position.

Discussion

Gornutz

duplicate of 27

Issue H-12: BlueBerryBank#withdrawLend will cause underlying token accounting error if soft/hard vault has withdraw fee

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/33

Found by

Ruhum, 0x52, evan, rvierdiiev, csanuragjain, y1cunhui

Summary

Soft/hard vaults can have a withdraw fee. This takes a certain percentage from the user when they withdraw. The way that the token accounting works in BlueBerryBank#withdrawLend, it will only remove the amount returned by the hard/soft vault from pos.underlying amount. If there is a withdraw fee, underlying amount will not be decrease properly and the user will be left with phantom collateral that they can still use.

Vulnerability Detail

```
// Cut withdraw fee if it is in withdrawVaultFee Window (2 months)
if (
    block.timestamp <
    config.withdrawVaultFeeWindowStartTime() +
        config.withdrawVaultFeeWindow()
) {
    uint256 fee = (withdrawAmount * config.withdrawVaultFee()) /
        DENOMINATOR;
    uToken.safeTransfer(config.treasury(), fee);
    withdrawAmount -= fee;
}
```

Both SoftVault and HardVault implement a withdraw fee. Here we see that withdrawAmount (the return value) is decreased by the fee amount.

```
uint256 wAmount;
if (address(ISoftVault(bank.softVault).uToken()) == token) {
    ISoftVault(bank.softVault).approve(
        bank.softVault,
        type(uint256).max
    );
    wAmount = ISoftVault(bank.softVault).withdraw(shareAmount);
} else {
```

```
    wAmount = IHardVault(bank.hardVault).withdraw(token, shareAmount);
}

wAmount = wAmount > pos.underlyingAmount
    ? pos.underlyingAmount
    : wAmount;

pos.underlyingVaultShare -= shareAmount;
pos.underlyingAmount -= wAmount;
bank.totalLend -= wAmount;
```

The return value is stored as `wAmount` which is then subtracted from `pos.underlyingAmount` the issue is that the withdraw fee has now caused a token accounting error for `pos`. We see that the fee paid to the hard/soft vault is NOT properly removed from `pos.underlyingAmount`. This leaves the user with phantom underlying which doesn't actually exist but that the user can use to take out loans.

Exmaple: For simplicity let's say that 1 share = 1 underlying and the soft/hard vault has a fee of 5%. Imagine a user deposits 100 underlying to receive 100 shares. Now the user withdraws their 100 shares while the hard/soft vault has a withdraw. This burns 100 shares and softVault/hardVault.withdraw returns 95 (100 - 5). During the token accounting pos.underlyingVaultShares are decreased to 0 but pos.underlyingAmount is still equal to 5 (100 - 95).

```
uint256 cv = oracle.getUnderlyingValue(
    pos.underlyingToken,
    pos.underlyingAmount
);
```

This accounting error is highly problematic because collateralValue uses pos.underlyingAmount to determine the value of collateral for liquidation purposes. This allows the user to take on more debt than they should.

Impact

User is left with collateral that isn't real but that can be used to take out a loan

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L669-L704

Tool used

Manual Review

Recommendation

`HardVault/SoftVault#withdraw` should also return the fee paid to the vault, so that it can be accounted for.

Issue H-13: IchiLpOracle is extemely easy to manipulate due to how IchiVault calculates underlying token balances

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/20

Found by

banditx0x, psy4n0n, obront, 0x52, carrot, cergyk, ctf_sec

Summary

`IchiVault#getTotalAmounts` uses the `UniV3Pool.slot0` to determine the number of tokens it has in it's position. `slot0` is the most recent data point and is therefore extremely easy to manipulate. Given that the protocol specializes in leverage, the effects of this manipulation would compound to make malicious uses even easier.

Vulnerability Detail

ICHIVault.sol

```
function _amountsForLiquidity(
    int24 tickLower,
    int24 tickUpper,
    uint128 liquidity
) internal view returns (uint256, uint256) {
    (uint160 sqrtRatioX96, , , , , , ) = IUniswapV3Pool(pool).slot0();
    return
        UV3Math.getAmountsForLiquidity(
            sqrtRatioX96,
            UV3Math.getSqrtRatioAtTick(tickLower),
            UV3Math.getSqrtRatioAtTick(tickUpper),
            liquidity
        );
}
```

`IchiVault#getTotalAmounts` uses the `UniV3Pool.slot0` to determine the number of tokens it has in it's position. slot0 is the most recent data point and can easily be manipulated.

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/oracle/IchiLpOracle.sol#L27-L36

`IchiLPOracle` directly uses the token values returned by `vault#getTotalAmounts`. This allows a malicious user to manipulate the valuation of the LP. An example of this kind of manipulation would be to use large buys/sells to alter the composition of the LP to make it worth less or more.

Impact

Ichi LP value can be manipulated to cause loss of funds for the protocol and other users

Code Snippet

Tool used

Manual Review

Recommendation

Token balances should be calculated inside the oracle instead of getting them from the `IchiVault`. To determine the liquidity, use a TWAP instead of `slot0`.

Issue H-14: WIchiFarm will break after second deposit of LP

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/15

Found by

0x52

Summary

WIchiFarm.sol makes the incorrect assumption that IchiVaultLP doesn't reduce allowance when using the transferFrom if allowance is set to type(uint256).max. Looking at a currently deployed IchiVault this assumption is not true. On the second deposit for the LP token, the call will always revert at the safe approve call.

Vulnerability Detail

IchiVault

```
function transferFrom(address sender, address recipient, uint256 amount) public
↪  virtual override returns (bool) {
    _transfer(sender, recipient, amount);
    _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount,
    ↪  "ERC20: transfer amount exceeds allowance"));
    return true;
}
```

The above lines show the trasnferFrom call which reduces the allowance of the spender regardless of whether the spender is approved for type(uint256).max or not.

```
if (
    IERC20Upgradeable(lpToken).allowance(
        address(this),
        address(ichiFarm)
    ) != type(uint256).max
) {
    // We only need to do this once per pool, as LP token's allowance won't
    ↪  decrease if it's -1.
    IERC20Upgradeable(lpToken).safeApprove(
        address(ichiFarm),
        type(uint256).max
    );
}
```

As a result after the first deposit the allowance will be less than type(uint256).max. When there is a second deposit, the reduced allowance will trigger a safeApprove call.

```
function safeApprove(
    IERC20Upgradeable token,
    address spender,
    uint256 value
) internal {
    // safeApprove should only be called when setting an initial allowance,
    // or when resetting it to zero. To increase and decrease it, use
    // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
    require(
        (value == 0) || (token.allowance(address(this), spender) == 0),
        "SafeERC20: approve from non-zero to non-zero allowance"
    );
    _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
    ↪  spender, value));
}
```

safeApprove requires that either the input is zero or the current allowance is zero. Since neither is true the call will revert. The result of this is that WIchiFarm is effectively broken after the first deposit.

Impact

WIchiFarm is broken and won't be able to process deposits after the first.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/wrapper/WIchiFarm.sol#L38

Tool used

Manual Review

Recommendation

Only approve is current allowance isn't enough for call. Optionally add zero approval before the approve. Realistically it's impossible to use the entire type(uint256).max, but to cover edge cases you may want to add it.

```
    if (
        IERC20Upgradeable(lpToken).allowance(
            address(this),
```

```
            address(ichiFarm)
-       ) != type(uint256).max
+       ) < amount
    ) {

+       IERC20Upgradeable(lpToken).safeApprove(
+           address(ichiFarm),
+           0
        );
        // We only need to do this once per pool, as LP token's allowance won't
        ↪  decrease if it's -1.
        IERC20Upgradeable(lpToken).safeApprove(
            address(ichiFarm),
            type(uint256).max
        );
    }
```

Issue M-1: The maximum size of an ICHI vault spell position can be arbitrarily surpassed

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/327

Found by

rvierdiiev, rbserver, berndartmueller, koxuan

Summary

The maximum size of an ICHI vault spell position can be arbitrarily surpassed by subsequent deposits to a position due to a flaw in the curPosSize calculation.

Vulnerability Detail

Ichi vault spell positions are subject to a maximum size limit to prevent large positions, ensuring a wide margin for liquidators and bad debt prevention for the protocol.

The maximum position size is enforced in the IchiVaultSpell.depositInternal function and compared to the current position size curPosSize.

However, the curPosSize does not reflect the actual position size, but the amount of Ichi vault LP tokens that are currently held in the IchiVaultSpell contract (see L153).

Assets can be repeatedly deposited into an Ichi vault spell position using the IchiVaultSpell.openPosition function (via the BlueBerryBank.execute function).

On the very first deposit, the curPosSize correctly reflects the position size. However, on subsequent deposits, the previously received Ichi vault LP tokens are kept in the BlueBerryBank contract. Thus, checking the balance of vault tokens in the IchiVaultSpell contract only accounts for the current deposit.

Test case

To demonstrate this issue, please use the following test case:

```
diff --git a/test/spell/ichivault.spell.test.ts
↪  b/test/spell/ichivault.spell.test.ts
index 258d653..551a6eb 100644
--- a/test/spell/ichivault.spell.test.ts
+++ b/test/spell/ichivault.spell.test.ts
@@ -163,6 +163,26 @@ describe('ICHI Angel Vaults Spell', () => {
                                afterTreasuryBalance.sub(beforeTreasuryBalance)
                        ).to.be.equal(depositAmount.mul(50).div(10000))
```

```
                })
+               it("should revert when exceeds max pos size due to increasing
↪  position", async () => {
+                       await ichi.approve(bank.address,
↪  ethers.constants.MaxUint256);
+                       await bank.execute(
+                               0,
+                               spell.address,
+                               iface.encodeFunctionData("openPosition", [
+                                       0, ICHI, USDC, depositAmount.mul(4),
↪  borrowAmount.mul(6) // Borrow 1.800e6 USDC
+                               ])
+                       );
+
+                       await expect(
+                               bank.execute(
+                                       0,
+                                       spell.address,
+                                       iface.encodeFunctionData("openPosition",
↪  [
+                                               0, ICHI, USDC,
↪  depositAmount.mul(1), borrowAmount.mul(2) // Borrow 300e6 USDC
+                                       ])
+                               )
+                       ).to.be.revertedWith("EXCEED_MAX_POS_SIZE"); // 1_800e6
↪  + 300e6 = 2_100e6 > 2_000e6 strategy max position size limit
+               })
                it("should be able to return position risk ratio", async () => {
                        let risk = await bank.getPositionRisk(1);
                        console.log('Prev Position Risk',
↪  utils.formatUnits(risk, 2), '%');
```

Run the test with the following command:

```
yarn hardhat test --grep "should revert when exceeds max pos size due to
↪  increasing position"
```

The test case fails and therefore shows that the maximum position size can be exceeded **without reverting**.

Impact

The maximum position size limit can be exceeded, leading to potential issues with liquidations and bad debt accumulation.

Code Snippet

spell/IchiVaultSpell.sol#L152-L156

```
122: function depositInternal(
123:     uint256 strategyId,
124:     address collToken,
125:     address borrowToken,
126:     uint256 collAmount,
127:     uint256 borrowAmount
128: ) internal {
...      // [...]
147:
148:     // 4. Validate MAX LTV
149:     _validateMaxLTV(strategyId);
150:
151:     // 5. Validate Max Pos Size
152:     uint256 lpPrice = bank.oracle().getPrice(strategy.vault);
153:     uint256 curPosSize = (lpPrice * vault.balanceOf(address(this))) /
154:         10**IICHIVault(strategy.vault).decimals();
155:     if (curPosSize > strategy.maxPositionSize)
156:         revert EXCEED_MAX_POS_SIZE(strategyId);
157: }
```

Tool used

Manual Review

Recommendation

Consider determining the current position size using the `bank.getPositionValue()` function instead of using the current Ichi vault LP token balance.

Issue M-2: Liquidations are enabled when repayments are disabled, causing borrowers to lose funds without a chance to repay

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/290

Found by

Jeiwan

Summary

Debt repaying can be temporary disabled by the admin of `BlueBerryBank`, however liquidations are not disabled during this period. As a result, users' positions can accumulate more borrow interest, go above the liquidation threshold, and be liquidated, while users aren't able to repay the debts.

Vulnerability Detail

The owner of BlueBerryBank can disable different functions of the contract, including repayments. However, while repayments are disabled liquidations are still allowed. As a result, when repayments are disabled, liquidator can liquidate any position, and borrowers won't be able to protect against that by repaying their debts. Thus, borrowers will be forced to lose their collateral.

Impact

Positions will be forced to liquidations while their owners won't be able to repay debts to avoid liquidations.

Code Snippet

BlueBerryBank.sol#L740

Tool used

Manual Review

Recommendation

Consider disallowing liquidations when repayments are disabled. Alternatively, consider never disallowing repayments so that users could maintain their positions in a healthy risk range anytime.

Issue M-3: IchiLpOracle returns inflated price due to invalid calculation

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/254

Found by

peanuts, tives, sakshamguruji

Summary

`IchiLpOracle` returns inflated price due to invalid calculation

Vulnerability Detail

If you run the tests, then you can see that IchiLpOracle returns inflated price for the ICHI_USDC vault

```
STATICCALL
↪  IchiLpOracle.getPrice(token=0xFCFE742e19790Dd67a627875ef8b45F17DB1DaC6) =>
↪  (1101189125194558706411110851447)
```

As the documentation says, the token price should be in USD with 18 decimals of precision. The price returned here is `1101189125194_558706411110851447` This is 1.1 trillion USD when considering the 18 decimals.

The test uses real values except for mocking ichi and usdc price, which are returned by the mock with correct decimals (1e18 and 1e6)

Impact

`IchiLpOracle` price is used in `_validateMaxLTV` (collToken is the vault). Therefore the collateral value is inflated and users can open bigger positions than their collateral would normally allow.

Code Snippet

```
/**
 * @notice Return lp token price in USD, with 18 decimals of precision.
 * @param token The underlying token address for which to get the price.
 * @return Price in USD
 */
function getPrice(address token) external view override returns (uint256) {
    IICHIVault vault = IICHIVault(token);
    uint256 totalSupply = vault.totalSupply();
```

```
    if (totalSupply == 0) return 0;

    address token0 = vault.token0();
    address token1 = vault.token1();

    (uint256 r0, uint256 r1) = vault.getTotalAmounts();
    uint256 px0 = base.getPrice(address(token0));
    uint256 px1 = base.getPrice(address(token1));
    uint256 t0Decimal = IERC20Metadata(token0).decimals();
    uint256 t1Decimal = IERC20Metadata(token1).decimals();

    uint256 totalReserve = (r0 * px0) /
        10**t0Decimal +
        (r1 * px1) /
        10**t1Decimal;

    return (totalReserve * 1e18) / totalSupply;
}
```

link

Tool used

Manual Review

Recommendation

Fix the LP token price calculation. The problem is that you multiply totalReserve with extra 1e18 (`return (totalReserve * 1e18) / totalSupply;`).

Discussion

Gornutz

duplicate of 15

Issue M-4: Use of deprecated `safeApprove()` is discouraged and missing approve to zero could cause certain token transfer to fail

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/236

Found by

RaymondFam, Jeiwan, PRAISE, Breeje, y1cunhui

Summary

OpenZeppelin's `safeapprove()` has issues similar to the ones found in IERC20.approve(), and its usage is discouraged.

Whenever possible, use safeIncreaseAllowance() and safeDecreaseAllowance() instead.

Additionally, some tokens, e.g USDT, do not work when changing the allowance from an existing non-zero allowance value. For instance, Tether (USDT)'s approve() will revert if the current approval has not been set to zero, serving to protect against front-running changes of approvals.

Vulnerability Detail

In the protocol, all functions using `safeapprove()` must be first approved by zero on top getting it replaced by `safeDecreaseAllowance()`. These are `ensureApprove()` in BasicSpell.sol, `initialize()` in SoftVault.sol, `mint()` and `burn()` in WichiFarm.sol.

Impact

Otherwise, these functions are going to revert every time they encounter such kind of tokens that might have a remaining allowance (even in dust amount) associated.

Code Snippet

File: BasicSpell.sol#L47-L52

```
function ensureApprove(address token, address spender) internal {
    if (!approved[token][spender]) {
        IERC20Upgradeable(token).safeApprove(spender, type(uint256).max);
        approved[token][spender] = true;
    }
}
```

File: SoftVault.sol#L41-L56

```
function initialize(
    IProtocolConfig _config,
    ICErc20 _cToken,
    string memory _name,
    string memory _symbol
) external initializer {
    __ERC20_init(_name, _symbol);
    __Ownable_init();
    if (address(_cToken) == address(0) || address(_config) == address(0))
        revert ZERO_ADDRESS();
    IERC20Upgradeable _uToken = IERC20Upgradeable(_cToken.underlying());
    config = _config;
    cToken = _cToken;
    uToken = _uToken;
    _uToken.safeApprove(address(_cToken), type(uint256).max);
}
```

File: WIchiFarm.sol#L82-L135

```
function mint(uint256 pid, uint256 amount)
    external
    nonReentrant
    returns (uint256)
{
    address lpToken = ichiFarm.lpToken(pid);
    IERC20Upgradeable(lpToken).safeTransferFrom(
        msg.sender,
        address(this),
        amount
    );
    if (
        IERC20Upgradeable(lpToken).allowance(
            address(this),
            address(ichiFarm)
        ) != type(uint256).max
    ) {
        // We only need to do this once per pool, as LP token's allowance won't
↪  decrease if it's -1.
        IERC20Upgradeable(lpToken).safeApprove(
            address(ichiFarm),
            type(uint256).max
        );
    }
    ichiFarm.deposit(pid, amount, address(this));
    (uint256 ichiPerShare, , ) = ichiFarm.poolInfo(pid);
```

```
    uint256 id = encodeId(pid, ichiPerShare);
    _mint(msg.sender, id, amount, "");
    return id;
}

function burn(uint256 id, uint256 amount)
    external
    nonReentrant
    returns (uint256)
{
    if (amount == type(uint256).max) {
        amount = balanceOf(msg.sender, id);
    }
    (uint256 pid, uint256 stIchiPerShare) = decodeId(id);
    _burn(msg.sender, id, amount);

    uint256 ichiRewards = ichiFarm.pendingIchi(pid, address(this));
    ichiFarm.harvest(pid, address(this));
    ichiFarm.withdraw(pid, amount, address(this));

    // Convert Legacy ICHI to ICHI v2
    if (ichiRewards > 0) {
        ICHIv1.safeApprove(address(ICHI), ichiRewards);
        ICHI.convertToV2(ichiRewards);
    }
```

Tool used

Manual Review

Recommendation

Consider approving 0 first prior to using the recommended `safeIncreaseAllowance()` to set the value of allowances.

For example, the first instance in the Code Snippet may be refactored as follows:

```
    function ensureApprove(address token, address spender) internal {
        if (!approved[token][spender]) {
+           IERC20Upgradeable(token).safeApprove(spender, 0);
+           IERC20Upgradeable(token).safeIncreaseAllowance(spender,
↪   type(uint256).max);
-           IERC20Upgradeable(token).safeApprove(spender, type(uint256).max);
            approved[token][spender] = true;
        }
    }
```

Issue M-5: potential gains of collateral are not given back to position owner when withdrawLend is called

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/227

Found by

Ch_301, koxuan

Summary

According to the docs, borrow tokens will bring potential gains or losses to position owners depending on the price of the collateral when closing a position. However, a logic in the `withdrawLend` function will only return position owner the original price that they have borrowed.

Vulnerability Detail

According to the docs https://docs.blueberry.garden/earn/what-are-liquidations,

So when would you have to worry about it? Well, since you've deposited crypto tokens, the value of your collateral is volatile and able to change as token prices move.

However, as we see in `withdrawLend`, if the withdrawn amount is more than the underlyingAmount, the potential gains are not given back to the position owner.

```
if (address(ISoftVault(bank.softVault).uToken()) == token) {
    ISoftVault(bank.softVault).approve(
        bank.softVault,
        type(uint256).max
    );
    wAmount = ISoftVault(bank.softVault).withdraw(shareAmount);
} else {
    wAmount = IHardVault(bank.hardVault).withdraw(token, shareAmount);
}


wAmount = wAmount > pos.underlyingAmount
    ? pos.underlyingAmount
    : wAmount;
```

Impact

Loss of fund to position owner when closing position.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L683-L695

Tool used

Manual Review

Recommendation

Recommend returning user their collateral and their potential gains.

Issue M-6: Safety net on computing total debt after borrowing

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/212

Found by

clems4ever

Summary

bank.totalDebt is updated when accrual is computed, borrowing is done or a repayment is made. In the case of accrual and repayment the actual value of the debt is coming from the cToken contract ensuring that the value is actual. When borrowing though, the value `amountCall` is added to the debt with no guarantee whether this amount has actually been borrowed.

I suggest that you align the computation of `totalDebt` with the other instances.

Vulnerability Detail

The value bank.totalDebt is updated with the value from the cToken in both cases below:

```
function doRepay(address token, uint256 amountCall)
        internal
        returns (uint256 repaidAmount)
    {
        Bank storage bank = banks[token]; // assume the input is already sanity
↪   checked.
        IERC20Upgradeable(token).approve(bank.cToken, amountCall);
        if (ICErc20(bank.cToken).repayBorrow(amountCall) != 0)
            revert REPAY_FAILED(amountCall);
        uint256 newDebt = ICErc20(bank.cToken).borrowBalanceStored(
↪   <=========================================
            address(this)
        );
        repaidAmount = bank.totalDebt - newDebt;
        bank.totalDebt = newDebt;
    }
```

```
function accrue(address token) public override {
        Bank storage bank = banks[token];
        if (!bank.isListed) revert BANK_NOT_LISTED(token);
        bank.totalDebt = ICErc20(bank.cToken).borrowBalanceCurrent(
↪   <=======================================
```

```
            address(this)
        );
    }
```

But in the case of a borrow the value is not coming from the cToken after the actual borrow as you can see in the snippet below

```
function doBorrow(address token, uint256 amountCall)
        internal
        returns (uint256 borrowAmount)
    {
        Bank storage bank = banks[token]; // assume the input is already sanity
↪   checked.

        IERC20Upgradeable uToken = IERC20Upgradeable(token);
        uint256 uBalanceBefore = uToken.balanceOf(address(this));
        if (ICErc20(bank.cToken).borrow(amountCall) != 0)
            revert BORROW_FAILED(amountCall);
        uint256 uBalanceAfter = uToken.balanceOf(address(this));

        borrowAmount = uBalanceAfter - uBalanceBefore;
        bank.totalDebt += amountCall;
↪   <==================================================
    }
```

This can be an issue in two cases:

1. The amount actually borrowed is less than amountCall, then the totalDebt will be recorded as bigger that what it actually is until the next accrual happens. In one case, the position might end up liquidatable and the operation execution will revert because of https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L607. or it can be not reverting and recording the wrong debt temporarily
2. The amount actually borrowed is bigger than amountCall (for instance because some fees might be accounted in the interests when entering the loan) and in that case the amount recorded into totalDebt might be less than what it is supposed to be until the next operation.

Impact

All view functions exposing the debt and the risk will not be accurate.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L868

Tool used

Manual Review

Recommendation

When borrowing, use the actual debt as seen by the cToken.

```
function doBorrow(address token, uint256 amountCall)
        internal
        returns (uint256 borrowAmount)
    {
        Bank storage bank = banks[token]; // assume the input is already sanity
↪   checked.

        IERC20Upgradeable uToken = IERC20Upgradeable(token);
        uint256 uBalanceBefore = uToken.balanceOf(address(this));
        if (ICErc20(bank.cToken).borrow(amountCall) != 0)
            revert BORROW_FAILED(amountCall);
        uint256 uBalanceAfter = uToken.balanceOf(address(this));

        borrowAmount = uBalanceAfter - uBalanceBefore;
        uint256 newDebt = ICErc20(bank.cToken).borrowBalanceStored(
            address(this)
        );
        bank.totalDebt += newDebt;
    }
```

Issue M-7: `BlueBerryBank.withdrawLend` function cannot be paused

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/204

Found by

rbserver

Summary

When an extreme market condition occurs, the protocol faces an exploit, or the authority issues a legal requirement, the protocol is able to pause the lending, borrowing, and repaying functionalities. However, the protocol is unable to pause the functionality for reducing a position.

Vulnerability Detail

Because the `BlueBerryBank` contract does not have a function, which is like `BlueBerryBank.isBorrowAllowed`, `BlueBerryBank.isRepayAllowed`, and `BlueBerryBank.isLendAllowed`, for pausing the functionality for reducing a position, the `BlueBerryBank.withdrawLend` function, which is shown in the Code Snippet section, cannot be paused. Users can still call the `IchiVaultSpell.reducePosition` function that further calls the `BlueBerryBank.withdrawLend` function to reduce a position when there is a need for pausing this functionality.

Impact

Just like pausing the lending, borrowing, and repaying functionalities, it is possible that the protocol needs to pause the functionality for reducing a position. However, when this need occurs, users can still reduce their positions through calling the `IchiVaultSpell.reducePosition` and `BlueBerryBank.withdrawLend` functions. The protocol cannot stop the outflow of the funds due to these position reductions even it is required to do so in this situation.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/IchiVaultSpell.sol#L266-L274

```
function reducePosition(
    uint256 strategyId,
    address collToken,
    uint256 collAmount
) external {
```

```
    doWithdraw(collToken, collAmount);
    doRefund(collToken);
    _validateMaxLTV(strategyId);
}
```

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L669-L704

```
function withdrawLend(address token, uint256 shareAmount)
    external
    override
    inExec
    poke(token)
{
    Position storage pos = positions[POSITION_ID];
    Bank storage bank = banks[token];
    if (token != pos.underlyingToken) revert INVALID_UTOKEN(token);
    if (shareAmount == type(uint256).max) {
        shareAmount = pos.underlyingVaultShare;
    }

    uint256 wAmount;
    if (address(ISoftVault(bank.softVault).uToken()) == token) {
        ISoftVault(bank.softVault).approve(
            bank.softVault,
            type(uint256).max
        );
        wAmount = ISoftVault(bank.softVault).withdraw(shareAmount);
    } else {
        wAmount = IHardVault(bank.hardVault).withdraw(token, shareAmount);
    }

    wAmount = wAmount > pos.underlyingAmount
        ? pos.underlyingAmount
        : wAmount;

    pos.underlyingVaultShare -= shareAmount;
    pos.underlyingAmount -= wAmount;
    bank.totalLend -= wAmount;

    wAmount = doCutWithdrawFee(token, wAmount);

    IERC20Upgradeable(token).safeTransfer(msg.sender, wAmount);
}
```

Tool used

Manual Review

Recommendation

A function, which is similar to the `BlueBerryBank.isBorrowAllowed`, `BlueBerryBank.isRepayAllowed`, and `BlueBerryBank.isLendAllowed` functions, can be added in the `BlueBerryBank` contract for pausing the `BlueBerryBank.withdrawLend` function. This function can then be used in the `BlueBerryBank.withdrawLend` function so the `BlueBerryBank.withdrawLend` function can be paused when needed.

Issue M-8: If a token's oracle goes down or price falls to zero, liquidations will be frozen

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/161

Found by

Saeedalipoor01988, 8olidity, obront, sakshamguruji, 0xChinedu

Summary

In some extreme cases, oracles can be taken offline or token prices can fall to zero. In these cases, liquidations will be frozen (all calls will revert) for any debt holders holding this token, even though they may be some of the most important times to allow liquidations to retain the solvency of the protocol.

Vulnerability Detail

Chainlink has taken oracles offline in extreme cases. For example, during the UST collapse, Chainlink paused the UST/ETH price oracle, to ensure that it wasn't providing inaccurate data to protocols.

In such a situation (or one in which the token's value falls to zero), all liquidations for users holding the frozen asset would revert. This is because any call to `liquidate()` calls `isLiquidatable()`, which calls `getPositionRisk()`, which calls the oracle to get the values of all the position's tokens (underlying, debt, and collateral).

Depending on the specifics, one of the following checks would cause the revert:

- the call to Chainlink's `registry.latestRoundData` would fail
- `if (updatedAt < block.timestamp - maxDelayTime) revert PRICE_OUTDATED(_token);`
- `if (px == 0) revert PRICE_FAILED(token);`

If the oracle price lookup reverts, liquidations will be frozen, and the user will be immune to liquidations. Although there are ways this could be manually fixed with fake oracles, by definition this happening would represent a cataclysmic time where liquidations need to be happening promptly to avoid the protocol falling into insolvency.

Impact

Liquidations may not be possible at a time when the protocol needs them most. As a result, the value of user's asset may fall below their debts, turning off any liquidation incentive and pushing the protocol into insolvency.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L511-L517

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L497-L505

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L477-L488

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/oracle/CoreOracle.sol#L182-L189

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/oracle/CoreOracle.sol#L95-L99

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/oracle/ChainlinkAdapterOracle.sol#L66-L84

Tool used

Manual Review

Recommendation

Ensure there is a safeguard in place to protect against this possibility.

Issue M-9: Incorrect handling of token approvals

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/159

Found by

8olidity, Bauer, carrot, shark, Breeje

Summary

The `BlueBerryBank` contract approves token spending. However, it never resets token approvals 0. This can cause issues with tokens like CRV, where you cannot set approvals from a non-zero value to another non-zero value.

Vulnerability Detail

Approvals are generally handled in two ways. In the normal Openzeppelin pattern, approve function sets the allowance to the value passed. However in other tokens, like USDT, there is a built-in race condition prevention mechanism where approvals must be reset to 0 before setting it to some non-zero value. This is also true for the case of the CRV token, as can be seen from its vyper contract

Since the protocol is intent on using CRV, it must use precaution with the approval mechanism and have the bank reset the approval to 0, after every instance of it granting approval. Otherwise, it can permanently brick the contract if even 1 wei of approval is remaining after calling the transfer functions, as it will revert on subsequent approval calls. Since the bank sometimes depends on external contracts to move out tokens (lending protocol, spells), the current approval pattern is very risky.

Impact

Bricked bank for CRV tokens due to incorrect approval handling

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L646-L657 https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L882-L884

Tool used

Manual Review

Recommendation

Explicitly reset approval to 0 after setting it at the end of the functions

```
IERC20Upgradeable(token).approve(bank.cToken, 0);
```

Issue M-10: totalLend isn't updated on liquidation, leading to permanently inflated value

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/155

Found by

SPYBOY, cducrest-brainbot, berndartmueller, obront

Summary

`bank.totalLend` tracks the total amount that has been lent of a given token, but it does not account for tokens that are withdrawn when a position is liquidated. As a result, the value will become overstated, leading to inaccurate data on the pool.

Vulnerability Detail

When a user lends a token to the Compound fork, the bank for that token increases its `totalLend` parameter:

```
bank.totalLend += amount;
```

Similarly, this value is decreased when the amount is withdrawn.

In the event that a position is liquidated, the `underlyingAmount` and `underlyingVaultShare` for the user are decreased based on the amount that will be transferred to the liquidator.

```
uint256 liqSize = (pos.collateralSize * share) / oldShare;
uint256 uTokenSize = (pos.underlyingAmount * share) / oldShare;
uint256 uVaultShare = (pos.underlyingVaultShare * share) / oldShare;

pos.collateralSize -= liqSize;
pos.underlyingAmount -= uTokenSize;
pos.underlyingVaultShare -= uVaultShare;
```

However, the liquidator doesn't receive those shares "inside the system". Instead, they receive the softVault tokens that can be claimed directly for the underlying asset by calling `withdraw()`, which simply redeems the underlying tokens from the Compound fork and sends them to the user.

```
function withdraw(uint256 shareAmount)
    external
    override
    nonReentrant
    returns (uint256 withdrawAmount)
```

```
{
    if (shareAmount == 0) revert ZERO_AMOUNT();

    _burn(msg.sender, shareAmount);

    uint256 uBalanceBefore = uToken.balanceOf(address(this));
    if (cToken.redeem(shareAmount) != 0) revert REDEEM_FAILED(shareAmount);
    uint256 uBalanceAfter = uToken.balanceOf(address(this));

    withdrawAmount = uBalanceAfter - uBalanceBefore;
    // Cut withdraw fee if it is in withdrawVaultFee Window (2 months)
    if (
        block.timestamp <
        config.withdrawVaultFeeWindowStartTime() +
            config.withdrawVaultFeeWindow()
    ) {
        uint256 fee = (withdrawAmount * config.withdrawVaultFee()) /
            DENOMINATOR;
        uToken.safeTransfer(config.treasury(), fee);
        withdrawAmount -= fee;
    }
    uToken.safeTransfer(msg.sender, withdrawAmount);

    emit Withdrawn(msg.sender, withdrawAmount, shareAmount);
}
```

Nowhere in this process is `bank.totalLend` updated. As a result, each time there is a liquidation of size X, `bank.totalLend` will move X higher relative to the correct value. Slowly, over time, this value will begin to dramatically misrepresent the accurate amount that has been lent.

While there is no material exploit based on this inaccuracy at the moment, this is a core piece of data in the protocol, and it's inaccuracy could lead to major issues down the road.

Furthermore, it will impact immediate user behavior, as the Blueberry devs have explained "we use that [value] to help us display TVL with subgraph", which will deceive and confuse users.

Impact

A core metric of the protocol will be permanently inaccurate, giving users incorrect data to make their assessments on and potentially causing more severe issues down the road.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L511-L572

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/vault/SoftVault.sol#L94-L123

Tool used

Manual Review

Recommendation

For the best accuracy, updating `bank.totalLend` should happen from the `withdraw()` function in `SoftVault.sol` instead of from the core `BlueberryBank.sol` contract.

Alternatively, you could add an update to `bank.totalLend` in the `liquidate()` function, which might temporarily underrepresent the total lent before the liquidator withdrew the funds, but would end up being accurate over the long run.

Issue M-11: Complete debt size is not paid off for fee on transfer tokens, but users aren't warned

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/153

Found by

chaduke, tsvetanovv, obront, rvierdiiev, Avci, Breeje, berndartmueller

Summary

The protocol seems to be intentionally catering to fee on transfer tokens by measuring token balances before and after transfers to determine the value received. However, the mechanism to pay the full debt will not succeed in paying off the debt if it is used with a fee on transfer token.

Vulnerability Detail

The protocol is clearly designed to ensure it is compatible with fee on transfer tokens. For example, all functions that receive tokens check the balance before and after, and calculate the difference between these values to measure tokens received:

```
function doERC20TransferIn(address token, uint256 amountCall)
    internal
    returns (uint256)
{
    uint256 balanceBefore = IERC20Upgradeable(token).balanceOf(
        address(this)
    );
    IERC20Upgradeable(token).safeTransferFrom(
        msg.sender,
        address(this),
        amountCall
    );
    uint256 balanceAfter = IERC20Upgradeable(token).balanceOf(
        address(this)
    );
    return balanceAfter - balanceBefore;
}
```

There is another feature of the protocol, which is that when loans are being repaid, the protocol gives the option of passing `type(uint256).max` to pay your debt in full:

```
if (amountCall == type(uint256).max) {
    amountCall = oldDebt;
```

```
}
```

However, these two features are not compatible. If a user paying off fee on transfer tokens passes in `type(uint256).max` to pay their debt in full, the full amount of their debt will be calculated. But when that amount is transferred to the contract, the amount that the result increases will be slightly less. As a result, the user will retain some balance that is not paid off.

Impact

The feature to allow loans to be paid in full will silently fail when used with fee on transfer tokens, which may trick users into thinking they have completely paid off their loans, and accidentally maintaining a balance.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L760-L775

Tool used

Manual Review

Recommendation

I understand that it would be difficult to implement a mechanism to pay fee on transfer tokens off in full. That adds a lot of complexity that is somewhat fragile.

The issue here is that the failure is silent, so that users request to pay off their loan in full, get confirmation, and may not realize that the loan still has an outstanding balance with interest accruing.

To solve this, there should be a confirmation that any user who passes `type(uint256).max` has paid off their debt in full. Otherwise, the function should revert, so that users paying fee on transfer tokens know that they cannot use the "pay in full" feature and must specify the correct amount to get their outstanding balance down to zero.

Issue M-12: If any incorrect parameters are added to a new bank, token will be permanently locked out of the protocol

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/149

Found by

GimelSec, obront

Summary

When new tokens are added to the protocol with the `addBank()` function, there aren't adequate checks on the parameters. These tokens are permanently added, many values are unchangeable, and the tokens added cannot be added again. The result is that any incorrect parameters will permanently lock these tokens from the platform.

Vulnerability Detail

When the admins add a new token to the protocol, they call the `addBank()` function:

```
function addBank(
    address token,
    address cToken,
    address softVault,
    address hardVault
    // @ok if oracle could stop supporting a token, it could still be added? -
↪   wont be accepted, admin error
) external onlyOwner onlyWhitelistedToken(token) {
    if (
        token == address(0) ||
        cToken == address(0) ||
        softVault == address(0) ||
        hardVault == address(0)
    ) revert ZERO_ADDRESS();
```

The function takes in the token address, cToken address, soft and hard vaults. The only logic check on these parameters is that they are not `address(0)`.

However, once these assets are set, the `token` and `cToken` are no longer allowed to be attached to a new bank.

```
if (cTokenInBank[cToken]) revert CTOKEN_ALREADY_ADDED();
if (bank.isListed) revert BANK_ALREADY_LISTED();
```

Similarly, the hard and soft vaults cannot be changed once they are set.

The result is that, if a token is added with an incorrect cToken, incorrect soft vault, or incorrect hard vault, these parameters cannot be changed, and the token cannot be edited, deleted or replaced, permanently blocking it from being used on the platform.

The same issue exists if, for any reason, the address of a cToken changed, or a token upgrades to a new address but uses the same cToken address, or any other changes occur that require these values to be shifted.

Note: I am aware that admin error is usually precluded from Sherlock contests, but this is not a "setup" task. It is an ongoing risk that could come from unlikely external events or from a small admin error, where any issue is unfixable and would cause major lasting damage. It is important that this possibility does not exist to ensure the protocol can function properly.

Impact

Important tokens could be locked from being allowed as an underlying or debt asset on the Blueberry protocol.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L190-L217

Tool used

Manual Review

Recommendation

Add a function for admins to update existing banks, changing the cToken, softVault or hardVault to ensure these markets are able to stay live and active.

Issue M-13: HardVault never deposits assets to Compound

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/147

Found by

obront, koxuan

Summary

While the protocol states that all underlying assets are deposited to their Compound fork to earn interest, it appears this action never happens in `HardVault.sol`.

Vulnerability Detail

The documentation and comments seem to make clear that all assets deposited to `HardVault.sol` should be deposited to Compound to earn yield:

```
/**
    * @notice Deposit underlying assets on Compound and issue share token
    * @param amount Underlying token amount to deposit
    * @return shareAmount cToken amount
    */
function deposit(address token, uint256 amount) { ... }

/**
    * @notice Withdraw underlying assets from Compound
    * @param shareAmount Amount of cTokens to redeem
    * @return withdrawAmount Amount of underlying assets withdrawn
    */
function withdraw(address token, uint256 shareAmount) { ... }
```

However, if we examine the code in these functions, there is no movement of the assets to Compound. Instead, they sit in the Hard Vault and doesn't earn any yield.

Impact

Users who may expect to be earning yield on their underlying tokens will not be.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/vault/HardVault.sol#L68-L116

Tool used

Manual Review

Recommendation

Either add the functionality to the Hard Vault to have the assets pulled from the ERC1155 and deposited to the Compound fork, or change the comments and docs to be clear that such underlying assets will not be receiving any yield.

Issue M-14: position owner can execute spell on position even if risk level has passed the liquidatable threshold.

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/133

Found by

koxuan

Summary

When executing spell, position will be checked to ensure that it is not in the liquidatable status. However, due to the design of the protocol to cache the `totalDebt` instead of querying compound forked protocol for the `totalDebt` everytime, cached `totalDebt` will not have the accrued interest from the point `execute` is called to the last `accrue` being called. User position might have entered liquidatable threshold but the cached `totalDebt` is not the latest and hence user can execute spell on position.

Vulnerability Detail

When user execute a spell on an existing position, a `isLiquiditable` check is run to make sure that user position has sufficient collateral.

```
function execute(
    uint256 positionId,
    address spell,
    bytes memory data
) external payable lock onlyEOAEx returns (uint256) {
    if (!whitelistedSpells[spell]) revert SPELL_NOT_WHITELISTED(spell);
    if (positionId == 0) {
        positionId = nextPositionId++;
        positions[positionId].owner = msg.sender;
    } else {
        if (positionId >= nextPositionId) revert BAD_POSITION(positionId);
        if (msg.sender != positions[positionId].owner)
            revert NOT_FROM_OWNER(positionId, msg.sender);
    }
    POSITION_ID = positionId;
    SPELL = spell;


    (bool ok, bytes memory returndata) = SPELL.call{value: msg.value}(data);
    if (!ok) {
        if (returndata.length > 0) {
            assembly {
```

```
                let returndata_size := mload(returndata)
                revert(add(32, returndata), returndata_size)
            }
        } else {
            revert("bad cast call");
        }
    }


    if (isLiquidatable(positionId)) revert INSUFFICIENT_COLLATERAL();


    POSITION_ID = _NO_ID;
    SPELL = _NO_ADDRESS;


    return positionId;
}
```

In `isLiquidatable`, position risk is used to determine if position has entered liquidatable status.

```
function isLiquidatable(uint256 positionId)
    public
    view
    returns (bool liquidatable)
{
    Position storage pos = positions[positionId];
    uint256 risk = getPositionRisk(positionId);
    liquidatable = risk >= oracle.getLiqThreshold(pos.underlyingToken);
}
```

```
function getPositionRisk(uint256 positionId)
    public
    view
    returns (uint256 risk)
{
    Position storage pos = positions[positionId];
    uint256 pv = getPositionValue(positionId);
    uint256 ov = getDebtValue(positionId);
    uint256 cv = oracle.getUnderlyingValue(
        pos.underlyingToken,
        pos.underlyingAmount
    );


    if (cv == 0) risk = 0;
```

```
    else if (pv >= ov) risk = 0;
    else {
        risk = ((ov - pv) * DENOMINATOR) / cv;
    }
}
```

`getDebtValue` in `getPositionRisk` is part of the formula to calculate the risk. Look at `uint256 debt = (share*bank.totalDebt).divCeil(bank.totalShare);` Notice that `bank.totalDebt`, the cached totalDebt of the bank is used to calculate position debt.

```
function getDebtValue(uint256 positionId)
    public
    view
    override
    returns (uint256)
{
    uint256 value = 0;
    Position storage pos = positions[positionId];
    uint256 bitMap = pos.debtMap;
    uint256 idx = 0;
    while (bitMap > 0) {
        if ((bitMap & 1) != 0) {
            address token = allBanks[idx];
            uint256 share = pos.debtShareOf[token];
            Bank storage bank = banks[token];
            uint256 debt = (share * bank.totalDebt).divCeil(
                bank.totalShare
            );
            value += oracle.getDebtValue(token, debt);
        }
        idx++;
        bitMap >>= 1;
    }
    return value;
}
```

If `accrue` is not called for a very long time due to inactivity for the token which means `poke` and `accrue` not being called, position debt will be lesser than the actual amount and thus owner can execute spell even if the current debt has passed the liquidatable threshold due to stale `totalDebt`.

Note: Other than `totalDebtValue`, upstream contracts that call view only functions like `getBankInfo`, `borrowBalanceStored` and `getPositionDebts` will get a lesser debt amount. If there are frontends or external contracts that rely on these functions, returning a stale value will cause upstream interfaces that rely on it to not work correctly.

Impact

Position owner can execute spell on position even if risk level has passed the liquidatable threshold.

Code Snippet

BlueBerryBank.sol#L578-L613 BlueBerryBank.sol#L497-L505
BlueBerryBank.sol#L477-L495 BlueBerryBank.sol#L451-L475

Tool used

Manual Review

Recommendation

Recommend calling compound forked protocol directly for the `totalDebt` if view only is needed for `getDebtValue` because compound protocol `borrowBalanceCurrent` is a view function unlike `poke` and `accrue`.

Issue M-15: Withdrawals from IchiVaultSpell have no slippage protection so can be frontrun, stealing all user funds

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/130

Found by

obront, 0x52, tives, koxuan, cergyk, rvierdiiev, ctf_sec, berndartmueller

Summary

When a user withdraws their position through the `IchiVaultSpell`, part of the unwinding process is to trade one of the released tokens for the other, so the borrow can be returned. This trade is done on Uniswap V3. The parameters are set in such a way that there is no slippage protection, so any MEV bot could see this transaction, aggressively sandwich attack it, and steal the majority of the user's funds.

Vulnerability Detail

Users who have used the `IchiVaultSpell` to take positions in Ichi will eventually choose to withdraw their funds. They can do this by calling `closePosition()` or `closePositionFarm()`, both of which call to `withdrawInternal()`, which follows loosely the following logic:

- sends the LP tokens back to the Ichi vault for the two underlying tokens (one of which was what was borrowed)
- swaps the non-borrowed token for the borrowed token on UniV3, to ensure we will be able to pay the loan back
- withdraw our underlying token from the Compound fork
- repay the borrow token loan to the Compound fork
- validate that we are still under the maxLTV for our strategy
- send the funds (borrow token and underlying token) back to the user

The issue exists in the swap, where Uniswap is called with the following function:

```
if (amountToSwap > 0) {
    swapPool = IUniswapV3Pool(vault.pool());
    swapPool.swap(
        address(this),
        !isTokenA,
        int256(amountToSwap),
        isTokenA
```

```
            ? UniV3WrappedLibMockup.MAX_SQRT_RATIO - 1
            : UniV3WrappedLibMockup.MIN_SQRT_RATIO + 1,
        abi.encode(address(this))
    );
}
```

The 4th variable is called `sqrtPriceLimitX96` and it represents the square root of the lowest or highest price that you are willing to perform the trade at. In this case, we've hardcoded in that we are willing to take the worst possible rate (highest price in the event we are trading 1 => 0; lowest price in the event we are trading 0 => 1).

The `IchiVaultSpell.sol#uniswapV3SwapCallback()` function doesn't enforce any additional checks. It simply sends whatever delta is requested directly to Uniswap.

```
function uniswapV3SwapCallback(
    int256 amount0Delta,
    int256 amount1Delta,
    bytes calldata data
) external override {
    if (msg.sender != address(swapPool)) revert NOT_FROM_UNIV3(msg.sender);
    address payer = abi.decode(data, (address));

    if (amount0Delta > 0) {
        if (payer == address(this)) {
            IERC20Upgradeable(swapPool.token0()).safeTransfer(
                msg.sender,
                uint256(amount0Delta)
            );
        } else {
            IERC20Upgradeable(swapPool.token0()).safeTransferFrom(
                payer,
                msg.sender,
                uint256(amount0Delta)
            );
        }
    } else if (amount1Delta > 0) {
        if (payer == address(this)) {
            IERC20Upgradeable(swapPool.token1()).safeTransfer(
                msg.sender,
                uint256(amount1Delta)
            );
        } else {
            IERC20Upgradeable(swapPool.token1()).safeTransferFrom(
                payer,
                msg.sender,
                uint256(amount1Delta)
            );
        }
```

```
    }
}
```

While it is true that there is an `amountRepay` parameter that is inputted by the user, it is not sufficient to protect users. Many users will want to make only a small repayment (or no repayment) while unwinding their position, and thus this variable will only act as slippage protection in the cases where users intend to repay all of their returned funds.

With this knowledge, a malicious MEV bot could watch for these transactions in the mempool. When it sees such a transaction, it could perform a "sandwich attack", trading massively in the same direction as the trade in advance of it to push the price out of whack, and then trading back after us, so that they end up pocketing a profit at our expense.

Because many of the ICHI token pairs have small amounts of liquidity (for example, ICHI-WBTC has under $350k), such an attack could feasible take the majority of the funds, leaving the user with close to nothing. See more details on liquidity here: https://info.uniswap.org/#/tokens/0x111111517e4929d3dcbdfa7cce55d30d4b6bc4d6

Impact

Users withdrawing their funds through the `IchiVaultSpell` who do not plan to repay all of the tokens returned from Uniswap could be sandwich attacked, losing their funds by receiving very little of their borrowed token back from the swap.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/IchiVaultSpell.sol#L300-L317

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/IchiVaultSpell.sol#L407-L442

Tool used

Manual Review

Recommendation

Have the user input a slippage parameter to ensure that the amount of borrowed token they receive back from Uniswap is in line with what they expect.

Alternatively, use the existing oracle system to estimate a fair price and use that value in the `swap()` call.

Issue M-16: Users lose part of `ICHI` rewards when they attempt to 'take' partial collateral by calling `closePositionFarm`

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/110

Found by

0Kage

Summary

Users can 'take' collateral by passing `lpTakeAmt` in `closePositionFarm`. `lpTakeAmt` in this case refers to wrapped farm tokens minted by `wIchiFarm` address. After taking back collateral from the bank, `wIchiFarm` calls the `burn()` function to harvest rewards and transfer them back to position owner.

`harvest` function of `ichiFarm` transfers the entire pending ITCHI rewards to the sender (which in this case is `wIchiFarm` address), regardless of the burn amount requested.

Of this, only proportionate amount of rewards are transfered back to the `ichiVaultSpell` - proportion here is determined by the `lpTakeAmt`. Balance tokens that rightfully belong to the position owner remain forvever trapped inside the `ichiVaultSpell` contract.

Vulnerability Detail

On sending a request to take partial collateral, `closePositionFarm` function calls the burn function on wIchiFarm. `burn()` calls the `harvest` function to collect all rewards from `wIchiFarm`.

On checking codebase of ichiFarm V2, we notice that `harvest` function transfers all rewards back to sender (which is `wIchiFarm`). Note that, even though we are burning partial tokens, `harvest` function returns rewards on the entire LP tokens present in farming contract.

```
function harvest(uint256 pid, address to) external {
    ...

      uint256 _pendingIchi = accumulatedIchi.sub(user.rewardDebt).toUInt256();

        // Effects
        user.rewardDebt = accumulatedIchi;

        // Interactions
        if (_pendingIchi > 0) {
            ICHI.safeTransfer(to, _pendingIchi);
```

```
        }

    ...

}
```

Now, coming back to the burn() function in wIchiFarm, notice that rewards only proportionate to burn amount are being transfered back to the `ichiVaultSpell` that will further transfer them to position owner.

```
function burn(uint256 id, uint256 amount)
       external
       nonReentrant
       returns (uint256)
   {

       ...

       ichiFarm.harvest(pid, address(this));

       ...

        // Transfer Reward Tokens
       (uint256 enIchiPerShare, , ) = ichiFarm.poolInfo(pid);
       uint256 stIchi = (stIchiPerShare * amount).divCeil(1e18);
       uint256 enIchi = (enIchiPerShare * amount) / 1e18;

       if (enIchi > stIchi) {
           ICHI.safeTransfer(msg.sender, enIchi - stIchi);
       }
       ...
   }
```

So the balance rewards are stuck inside the spell contract & there is no way for users to get back the rewards on subsequent burning of farm tokens.

Impact

Users taking partial collateral will lose part of their rewards permanently.

POC

- Alice is an existing borrower with following positions (1 ETH - underlying, 700 USDC borrowing, collateral: 400 ICHI farm tokens)
- Alice requests to take back 100 ICHI farm tokens

- 100 ICHI farm tokens are burnt - and say all 400 ICHI farm tokens have collected 30 ICHI reward tokens
- `wIchiFarm` wrapper gets back all 30 ICHI rewards
- Wrapper sends back only 7.5 ICHI rewards (25% of total rewards corresponding to 25% of ICHI farm tokens burnt) to `ichiVaultSpell`
- `ichiVaultSpell` sends back 7.5 ICHI rewards to Alice
- Alice loses balance 22.5 ICHI rewards which are permanently trapped inside the `wIchiFarm` wrapper

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/wrapper/WIchiFarm.sol#L128

Tool used

Manual Review

Recommendation

Transfer back all rewards to `ichiVaultSpell` which inturn transfers them back to the owner.

Inside the burn() function, replace code block where rewards propotionate to `amount` are calculated....

```
(uint256 enIchiPerShare, , ) = ichiFarm.poolInfo(pid);
uint256 stIchi = (stIchiPerShare * amount).divCeil(1e18);
uint256 enIchi = (enIchiPerShare * amount) / 1e18;

if (enIchi > stIchi) {
    ICHI.safeTransfer(msg.sender, enIchi - stIchi);
}
```

... with `ichiRewards` instead - this way all rewards upto that point are sent back to owner

```
ICHI.safeTransfer(msg.sender, ichiRewards); //**** - audit - ichiRewards are the
↪  entire pending rewards upto this point****
```

Issue M-17: BlueBerryBank.getPositionRisk shows risk 0, when underlying price is 0

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/104

Found by

rvierdiiev

Summary

BlueBerryBank.getPositionRisk shows risk 0, when underlying price is 0

Vulnerability Detail

BlueBerryBank.getPositionRisk function is responsible for providing risk of position. Depending on this risk position can be liquidated. https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L477-L495

```
function getPositionRisk(uint256 positionId)
    public
    view
    returns (uint256 risk)
{
    Position storage pos = positions[positionId];
    uint256 pv = getPositionValue(positionId);
    uint256 ov = getDebtValue(positionId);
    uint256 cv = oracle.getUnderlyingValue(
        pos.underlyingToken,
        pos.underlyingAmount
    );


    if (cv == 0) risk = 0;
    else if (pv >= ov) risk = 0;
    else {
        risk = ((ov - pv) * DENOMINATOR) / cv;
    }
}
```

The problem that in case if cv == 0, returned risk is 0. If cv == 0 that means that price of underlying token became 0 and that risk should be maximum.

1.Suppose that user borrowed using some underlying token tokenA. 2.Then smth happened with that token, so it costs nothing now. 3.But position is not liquidatable as it will return risk 0 and is considered as healthy.

Impact

Lost of borrowed funds for protocol.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L477-L495

Tool used

Manual Review

Recommendation

In case if underlying token price became 0 you need to return risk that is already liquidatable.

Issue M-18: Chainlink's latestRoundData return stale or incorrect result

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/94

Found by

SPYBOY, tsvetanovv, 8olidity, Nyx, HonorLt, obront, evan, 0x52, Aymen0909, hl_, koxuan, Chinmay, peanuts, csanuragjain, WatchDogs, rbserver, Avci

Summary

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/oracle/ChainlinkAdapterOracle.sol#L76

Vulnerability Detail

Impact

On ChainlinkAdapterOracle.sol, you are using latestRoundData, but there is no check if the return value indicates stale data.

```
function getPrice(address _token) external view override returns (uint256) {
        // remap token if possible
        address token = remappedTokens[_token];
        if (token == address(0)) token = _token;

        uint256 maxDelayTime = maxDelayTimes[token];
        if (maxDelayTime == 0) revert NO_MAX_DELAY(_token);

        // try to get token-USD price
        uint256 decimals = registry.decimals(token, USD);
        (, int256 answer, , uint256 updatedAt, ) = registry.latestRoundData(
            token,
            USD
        );
        if (updatedAt < block.timestamp - maxDelayTime)
            revert PRICE_OUTDATED(_token);

        return (answer.toUint256() * 1e18) / 10**decimals;
    }
```

This could lead to stale prices according to the Chainlink documentation: https://docs.chain.link/data-feeds/price-feeds/historical-data Related report: https://github.com/code-423n4/2021-05-fairside-findings/issues/70

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/oracle/ChainlinkAdapterOracle.sol#L76

Tool used

Manual Review

Recommendation

Add the below check for returned data

```
function getPrice(address _token) external view override returns (uint256) {
        // remap token if possible
        address token = remappedTokens[_token];
        if (token == address(0)) token = _token;

        uint256 maxDelayTime = maxDelayTimes[token];
        if (maxDelayTime == 0) revert NO_MAX_DELAY(_token);

        // try to get token-USD price
        uint256 decimals = registry.decimals(token, USD);
        (uint80 roundID, int256 answer, uint256 timestamp, uint256 updatedAt, )
↪   = registry.latestRoundData(
            token,
            USD
        );
        //Solution
        require(updatedAt >= roundID, "Stale price");
        require(timestamp != 0,"Round not complete");
        require(answer > 0,"Chainlink answer reporting 0");

        if (updatedAt < block.timestamp - maxDelayTime)
            revert PRICE_OUTDATED(_token);

        return (answer.toUint256() * 1e18) / 10**decimals;
    }
```

Issue M-19: BasicSpell.doCutRewardsFee uses depositFee instead of withdraw fee

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/82

Found by

rvierdiiev

Summary

BasicSpell.doCutRewardsFee uses depositFee instead of withdraw fee

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/BasicSpell.sol#L65-L79

```
function doCutRewardsFee(address token) internal {
    if (bank.config().treasury() == address(0)) revert NO_TREASURY_SET();


    uint256 balance = IERC20Upgradeable(token).balanceOf(address(this));
    if (balance > 0) {
        uint256 fee = (balance * bank.config().depositFee()) / DENOMINATOR;
        IERC20Upgradeable(token).safeTransfer(
            bank.config().treasury(),
            fee
        );


        balance -= fee;
        IERC20Upgradeable(token).safeTransfer(bank.EXECUTOR(), balance);
    }
}
```

This function is called in order to get fee from ICHI rewards, collected by farming. But currently it takes `bank.config().depositFee()` instead of `bank.config().withdrawFee()`.

Impact

Wrong fee amount is taken.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/BasicSpell.sol#L65-L79

Tool used

Manual Review

Recommendation

Take withdraw fee from rewards.

Issue M-20: A Whale can grieve specific type of borrowing token

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/80

Found by

mert_eren

Summary

A Whale can grieve specific type of borrowing token

Vulnerability Detail

ichiVault token taken with amount of borrowed token in spell contract balance. And total IchiVault token value houldn't be exceed the strategies set value. However, a malicious user can transfer borrow token and cause a revert of other user's openPosition attempt even if their borrowToken amount not exceed the limit.

Impact

https://imgur.com/a/XH8NAr6 This test work in ichiVaultSpell.ts As can seen in photo even if a normal user's openPosiiton attempt revert due to contract try to take much more vault token than the user's borrowAmount.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/IchiVaultSpell.sol#L140-L146

Tool used

Manual Review

Recommendation

Try to deposit user's borrowAmount input instead of borrowToken.balanceOf(address(this))

Issue M-21: A whale can make grieving attack to strategies that use same vault by transfer vault token to spell.

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/79

Found by

mert_eren

Summary

A whale can make grieving attack to strategies that use same vault by transfer vault token to spell.

Vulnerability Detail

In ichiVaultSpell's depositInternal function it check total vault token in spell contract. If totalVaultToken*vaultPrice exceed strategies maxPosition size than reverts. So a whale can take large amount of vault token and transfer to the spell contract and can make a dos of openposition for strategies which use this specific vault.

Impact

https://imgur.com/a/6C7Gagw This test work in ichiVaultSpellTest.ts As seen normal user can't openPosition and take EXCEED_MAX_POS warning even if it shouldn't.

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/spell/IchiVaultSpell.sol#L153-L157

Tool used

Manual Review

Recommendation

Instead of using balanceOf function can be recorded within the contract.

Issue M-22: A borrower might drain the vault by calling borrow() repeatedly with small borrow amount each time.

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/45

Found by

chaduke

Summary

A borrower might drain the vault by calling `borrow()` repeatedly with small borrow amount that converts to a zero debt share each time.

Vulnerability Detail

This is possible because `borrow()` does not check whether the number of shares borrowed is equal to zero or not. Therefore, an attacker can take advantage of the rounding error and borrow funds for free. We show how a borrowed can drain the vault by calling borrow() repeatedly:

1) Suppose the for a particular token X, the total bank debt is 1000,000 and the total debt share is 100,000. That is each debt share has a 10 debt.
2) A malicious borrower Bob can call `borrow()` (via SPELL) and borrow 9 each time, which will convert to `9*100,000/1000,000 = 0` debt shares.

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L709-L735

3) As a result, the borrower can steal 9 X tokens each time the `borrow()` is called without increasing his debt shares. Bob can call this repeatedly in one transaction `Steal()` (within the gas limit) to borrow many tokens of X without increasing any debt shares.
4) Call `Steal()` many times, Bob will be able to drain the vault.

Impact

A malicious borrower can drain the the vault by calling `borrow()` repeatedly.

Code Snippet

See above

Tool used

VScode Manual Review

Recommendation

Borrow should revert when `newShare == 0`.

```
 function borrow(address token, uint256 amount)
        external
        override
        inExec
        poke(token)
        onlyWhitelistedToken(token)
    {
        if (!isBorrowAllowed()) revert BORROW_NOT_ALLOWED();
        Bank storage bank = banks[token];
        Position storage pos = positions[POSITION_ID];
        uint256 totalShare = bank.totalShare;
        uint256 totalDebt = bank.totalDebt;
        uint256 share = totalShare == 0
            ? amount
            : (amount * totalShare).divCeil(totalDebt);
+       if(share == 0) revert BorrowZeroShare();

        bank.totalShare += share;
        uint256 newShare = pos.debtShareOf[token] + share;
        pos.debtShareOf[token] = newShare;
        if (newShare > 0) {
            pos.debtMap |= (1 << uint256(bank.index));
        }
        IERC20Upgradeable(token).safeTransfer(
            msg.sender,
            doBorrow(token, amount)
        );
        emit Borrow(POSITION_ID, msg.sender, token, amount, share);
    }
```

Issue M-23: onlyEOAEx modifier that ensures call is from EOA might not hold true in the future

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/21

Found by

koxuan

Summary

modifier `onlyEOAEx` is used to ensure calls are only made from EOA. However, EIP 3074 suggests that using `onlyEOAEx` modifier to ensure calls are only from EOA might not hold true.

Vulnerability Detail

For `onlyEOAEx`, `tx.origin` is used to ensure that the caller is from an EOA and not a smart contract.

```
modifier onlyEOAEx() {
    if (!allowContractCalls && !whitelistedContracts[msg.sender]) {
        if (msg.sender != tx.origin) revert NOT_EOA(msg.sender);
    }
    _;
}
```

However, according to EIP 3074,

This EIP introduces two EVM instructions AUTH and AUTHCALL. The first sets a context variable authorized based on an ECDSA signature. The second sends a call as the authorized account. This essentially delegates control of the externally owned account (EOA) to a smart contract.

Therefore, using tx.origin to ensure msg.sender is an EOA will not hold true in the event EIP 3074 goes through.

Impact

Using modifier `onlyEOAEx` to ensure calls are made only from EOA will not hold true in the event EIP 3074 goes through.

Code Snippet

BlueBerryBank.sol#L54-L59

Tool used

Manual Review

Recommendation

Recommend using OpenZepellin's `isContract` function (https://docs.openzeppelin.com/contracts/2.x/api/utils#Address-isContract-address-). Note that there are edge cases like contract in constructor that can bypass this and hence caution is required when using this.

```
modifier onlyEOAEx() {
    if (!allowContractCalls && !whitelistedContracts[msg.sender]) {
        if (isContract(msg.sender)) revert NOT_EOA(msg.sender);
    }
    _;
}
```

Issue M-24: liquidate will revert if amountCall is more than debt which can lead to DOS

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/14

Found by

stent, tsvetanovv, koxuan

Summary

A user can liquidate a liquidatable position. However, a logic in the coding allows others to frontrun liquidate tx and DOS it.

Vulnerability Detail

Let's say a position has 1000 debt. A user sees that the position is liquidatable, and calls liquidate with the full amount of 1000 debtToken.

```
function liquidate(
    uint256 positionId,
    address debtToken,
    uint256 amountCall
) external override lock poke(debtToken) {
    if (amountCall == 0) revert ZERO_AMOUNT();
    if (!isLiquidatable(positionId)) revert NOT_LIQUIDATABLE(positionId);
    Position storage pos = positions[positionId];
    Bank memory bank = banks[pos.underlyingToken];
    if (pos.collToken == address(0)) revert BAD_COLLATERAL(positionId);

    uint256 oldShare = pos.debtShareOf[debtToken];
    (uint256 amountPaid, uint256 share) = repayInternal(
        positionId,
        debtToken,
        amountCall
    );

    uint256 liqSize = (pos.collateralSize * share) / oldShare;
    uint256 uTokenSize = (pos.underlyingAmount * share) / oldShare;
    uint256 uVaultShare = (pos.underlyingVaultShare * share) / oldShare;

    pos.collateralSize -= liqSize;
    pos.underlyingAmount -= uTokenSize;
    pos.underlyingVaultShare -= uVaultShare;

    // Transfer position (Wrapped LP Tokens) to liquidator
```

```
    IERC1155Upgradeable(pos.collToken).safeTransferFrom(
        address(this),
        msg.sender,
        pos.collId,
        liqSize,
        ""
    );
    // Transfer underlying collaterals(vault share tokens) to liquidator
    if (
        address(ISoftVault(bank.softVault).uToken()) == pos.underlyingToken
    ) {
        IERC20Upgradeable(bank.softVault).safeTransfer(
            msg.sender,
            uVaultShare
        );
    } else {
        IERC1155Upgradeable(bank.hardVault).safeTransferFrom(
            address(this),
            msg.sender,
            uint256(uint160(pos.underlyingToken)),
            uVaultShare,
            ""
        );
    }

    emit Liquidate(
        positionId,
        msg.sender,
        debtToken,
        amountPaid,
        share,
        liqSize,
        uTokenSize
    );
}
```

A griefer sees the tx in the mempool. He decides to frontrun the tx by calling `liquidate` with 1 amountCall. Now the debt would be 999. Now notice that in `repayInternal` that if `paid > oldDebt`, the tx will revert. User can keep frontrunning liquidator and preventing him from repaying the full amount.

```
function repayInternal(
    uint256 positionId,
    address token,
    uint256 amountCall
) internal returns (uint256, uint256) {
```

```
    Bank storage bank = banks[token];
    Position storage pos = positions[positionId];
    uint256 totalShare = bank.totalShare;
    uint256 totalDebt = bank.totalDebt;
    uint256 oldShare = pos.debtShareOf[token];
    uint256 oldDebt = (oldShare * totalDebt).divCeil(totalShare);
    if (amountCall == type(uint256).max) {
        amountCall = oldDebt;
    }
    amountCall = doERC20TransferIn(token, amountCall);
    uint256 paid = doRepay(token, amountCall);
    if (paid > oldDebt) revert REPAY_EXCEEDS_DEBT(paid, oldDebt); // prevent
↪   share overflow attack
    uint256 lessShare = paid == oldDebt
        ? oldShare
        : (paid * totalShare) / totalDebt;
    bank.totalShare = totalShare - lessShare;
    uint256 newShare = oldShare - lessShare;
    pos.debtShareOf[token] = newShare;
    if (newShare == 0) {
        pos.debtMap &= ~(1 << uint256(bank.index));
    }
    return (paid, lessShare);
}
```

```
function doRepay(address token, uint256 amountCall)
    internal
    returns (uint256 repaidAmount)
{
    Bank storage bank = banks[token]; // assume the input is already sanity
↪   checked.
    IERC20Upgradeable(token).approve(bank.cToken, amountCall);
    if (ICErc20(bank.cToken).repayBorrow(amountCall) != 0)
        revert REPAY_FAILED(amountCall);
    uint256 newDebt = ICErc20(bank.cToken).borrowBalanceStored(
        address(this)
    );
    repaidAmount = bank.totalDebt - newDebt;
    bank.totalDebt = newDebt;
}
```

Impact

Adversary can cause DOS to `liquidate` by preventing liquidator from calling `liquidate` with full amount.

Code Snippet

BlueBerryBank.sol#L511-L572 BlueBerryBank.sol#L760-L787 BlueBerryBank.sol#L877-L890

Tool used

Manual Review

Recommendation

If amountCall is more than debt, it should be set to debt

```
if (amountCall > oldDebt){
  amountCall = oldDebt;
}
```

Issue M-25: BlueBerryBank#doCutDepositFee is problematic for ERC20 tokens that don't support zero transfers

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/13

Found by

0x52, peanuts, rvierdiiev

Summary

BlueBerryBank#doCutDepositFee attempts to transfer funds even if there isn't a deposit fee. If the underlying ERC20 doens't support transfers with zero value then the call will always revert when the deposit fee is zero.

Vulnerability Detail

```
function doCutDepositFee(address token, uint256 amount)
    internal
    returns (uint256)
{
    if (config.treasury() == address(0)) revert NO_TREASURY_SET();
    uint256 fee = (amount * config.depositFee()) / DENOMINATOR;
    IERC20Upgradeable(token).safeTransfer(config.treasury(), fee);
    return amount - fee;
}
```

BlueBerryBank#doCutDepositFee always calls safeTransfer even when the amount to send could be zero because there isn't any deposit fee. For ERC20 tokens that don't support zero value transfers, this will always revert which breaks support for them.

Impact

Having no deposit fee will break support for ERC20 tokens that don't support zero transfers

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/BlueBerryBank.sol#L892-L900

Tool used

Manual Review

Recommendation

Only transfer if the amount to transfer is not zero:

```
function doCutDepositFee(address token, uint256 amount)
    internal
    returns (uint256)
{
    if (config.treasury() == address(0)) revert NO_TREASURY_SET();
    uint256 fee = (amount * config.depositFee()) / DENOMINATOR;
-   IERC20Upgradeable(token).safeTransfer(config.treasury(), fee);
+   if (fee != 0) IERC20Upgradeable(token).safeTransfer(config.treasury(), fee);
    return amount - fee;
}
```

Issue M-26: ChainlinkAdapterOracle use BTC/USD chainlink oracle to price WBTC which is problematic if WBTC depegs

Source: https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/9

Found by

0x52

Summary

The chainlink BTC/USD oracle is used to price WBTC (docs). WBTC is basically a bridged asset and if the bridge is compromised/fails then WBTC will depeg and will no longer be equivalent to BTC. This will lead to large amounts of borrowing against an asset that is now effectively worthless. Since the protocol still values it via BTC/USD the protocol will not only be stuck with the bad debt caused by the currently outstanding loans but they will also continue to give out bad loans and increase the amount of bad debt further

Vulnerability Detail

See summary.

Impact

Protocol will take on a large amount of bad debt should WBTC bridge become compromised and WBTC depegs

Code Snippet

https://github.com/sherlock-audit/2023-02-blueberry/blob/main/contracts/oracle/ChainlinkAdapterOracle.sol#L47-L59

Tool used

Manual Review

Recommendation

I would recommend using a double oracle setup. Use both the Chainlink and another on-chain liquidity base oracle (i.e. UniV3 TWAP). If the price of the on-chain liquidity oracle drops below a certain threshold of the Chainlink oracles (i.e. 2% lower), any borrowing should be immediately halted. The chainlink oracle

will prevent price manipulation and the liquidity oracle will safeguard against the asset depegging.