SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Isomorph
Prepared by:	Sherlock
Lead Security Expert:	0x52
Dates Audited:	November 23 - December 7, 2022
Prepared on:	December 24, 2022

Introduction

Isomorph is an Optimism-native lending protocol with a focus on novel interest generating collaterals. Loans are issued through a minted stablecoin isoUSD.

Scope

The following contracts in the repo Isomorph @789338c are in scope:

- `CollateralBook.sol`
- `Locker.sol`
- `RoleControl.sol`
- `Vault_Base_ERC20.sol`
- `Vault_Lyra.sol`
- `Vault_Synths.sol`
- `Vault_Velo.sol`
- `isoUSDToken.sol`

Also included as the following contracts contained in the Velo-Deposit-Tokens @1e92342 submodule

- `DepositReceipt_Base.sol`
- `DepositReceipt_ETH.sol`
- `DepositReceipt_USDC.sol`
- `Depositor.sol`
- `Templater.sol`

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
13	9

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0x52
clems4ever
rvierdiiev
HollaDieWaldfee
HonorLt
hansfriese
ak1
cccz
ctf_sec
Jeiwan
CodingNameKiki
bin2chen
yixxas
caventa
libratus
KingNFT
Deivitto
neumo
rotcivegaf
GimelSec
__141345__
8olidity
wagmi
0xjayne
0x4non
jonatascm
Atarpara

Issue H-1: User is unable to partially payback loan if they aren't able to post enough isoUSD to bring them back to minOpeningMargin

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/161

Found by

0x52

Summary

The only way for a user to reduce their debt is to call closeLoan. If the amount repaid does not bring the user back above minOpeningMargin then the transaction will revert. This is problematic for users that wish to repay their debt but don't have enough to get back to minOpeningMargin as it could lead to unfair liquidations.

Vulnerability Detail

```
if(outstandingisoUSD >= TENTH_OF_CENT){ //ignore leftover debts less than $0.001
    uint256 collateralLeft = collateralPosted[_collateralAddress][msg.sender] -
    ↪  _collateralToUser;
    uint256 colInUSD = priceCollateralToUSD(currencyKey, collateralLeft);
    uint256 borrowMargin = (outstandingisoUSD * minOpeningMargin) / LOAN_SCALE;
    require(colInUSD > borrowMargin , "Remaining debt fails to meet minimum
    ↪  margin!");
}
```

The checks above are done when a user calls closeLoan. This ensures that the user's margin is back above minOpeningMargin before allowing them to remove any collateral. This is done as a safeguard to block loans users from effectively opening loans at lower than desired margin. This has the unintended consequence that as user cannot pay off any of their loan if they do not increase their loan back above minOpeningMargin. This could prevent users from being able to save a loan that is close to liquidation causing them to get liquidated when they otherwise would have paid off their loan.

Impact

User is unable to make partial repayments if their payment does not increase margin enough

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Synths.sol#L197-L248

Tool used

Manual Review

Recommendation

I recommend adding a separate function that allows users to pay off their loan without removing any collateral:

```
function paybackLoan(
    address _collateralAddress,
    uint256 _USDToVault
    ) external override whenNotPaused
    {
    _collateralExists(_collateralAddress);
    _closeLoanChecks(_collateralAddress, 0, _USDToVault);
    //make sure virtual price is related to current time before fetching
    ↪  collateral details
    //slither-disable-next-line reentrancy-vulnerabilities-1
    _updateVirtualPrice(block.timestamp, _collateralAddress);
    (
        bytes32 currencyKey,
        uint256 minOpeningMargin,
        ,
        ,
        ,
        uint256 virtualPrice,

    ) = _getCollateral(_collateralAddress);
    //check for frozen or paused collateral
    _checkIfCollateralIsActive(currencyKey);

    uint256 isoUSDdebt = (isoUSDLoanAndInterest[_collateralAddress][msg.sender]
    ↪  * virtualPrice) / LOAN_SCALE;
    require( isoUSDdebt >= _USDToVault, "Trying to return more isoUSD than
    ↪  borrowed!");
    uint256 outstandingisoUSD = isoUSDdebt - _USDToVault;

    uint256 collateral = collateralPosted[_collateralAddress][msg.sender];
    uint256 colInUSD = priceCollateralToUSD(currencyKey, collateral);
    uint256 borrowMargin = (outstandingisoUSD * liquidatableMargin) / LOAN_SCALE;
    require(colInUSD > borrowMargin , "Liquidation margin not met!");
```

```
    //record paying off loan principle before interest
    //slither-disable-next-line uninitialized-local-variables
    uint256 interestPaid;
    uint256 loanPrinciple = isoUSDLoaned[_collateralAddress][msg.sender];
    if( loanPrinciple >= _USDToVault){
        //pay off loan principle first
        isoUSDLoaned[_collateralAddress][msg.sender] = loanPrinciple -
        ↪  _USDToVault;
    }
    else{
        interestPaid = _USDToVault - loanPrinciple;
        //loan principle is fully repaid so record this.
        isoUSDLoaned[_collateralAddress][msg.sender] = 0;
    }
    //update mappings with reduced amounts
    isoUSDLoanAndInterest[_collateralAddress][msg.sender] =
    ↪  isoUSDLoanAndInterest[_collateralAddress][msg.sender] - ((_USDToVault *
    ↪  LOAN_SCALE) / virtualPrice);
    emit ClosedLoan(msg.sender, _USDToVault, currencyKey, 0);
    //Now all effects are handled, transfer the assets so we follow CEI pattern
    _decreaseLoan(_collateralAddress, 0, _USDToVault, interestPaid);
}
```

Discussion

kree-dotcom

Sponsor confirmed. This is a difficult fix, it is highly likely that adding an extra function to Vault_Lyra and Vault_Velo will lead to "code too big" errors preventing them compiling, we will have to consult with 0x52/Sherlock to see what else can be done to fix this.

kree-dotcom

Proposing to change the `closeLoan()` check to

`if((outstandingisoUSD > 0) && (_collateralToUser > 0)){ //leftover debt must meet minOpeningMargin if requesting collateral back uint256 collateralLeft = ... ... }`

This way checks are only triggered if the user is repaying some of their loan and requesting capital back. This allows us to prevent people from opening loans with a lower than minOpeningMargin but allows them to reduce their loan without needing to match the minOpeningMargin.

kree-dotcom

Applied fix to `Vault_Synth` here https://github.com/kree-dotcom/isomorph/commit/

6b0879e59a3e5156fde88f4c90d0029aa31b3786 `Vault_Velo` and `Vault_Lyra` fixed here https://github.com/kree-dotcom/isomorph/commit/7b113eaf19c126db796190f271d85bc5c2cba865

Note `Vault_Velo` has to use a slightly different method because we are handling NFTs instead of ERC20s. I have checked the `_calculateProposedReturnedCapital()` function we are relying on will also work fine if given an array of non-owned NFTs (i.e. the user is not receiving any collateral back just reducing their loan) by returning 0.

Fix typo in Vault_Velo https://github.com/kree-dotcom/isomorph/commit/67b2f981a7cbe3fb0b6a057dab82d69d6d61fdc9

Issue H-2: The calculation of `totalUSDborrowed` in `openLoan()` is not correct

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/85

Found by

libratus, cccz, clems4ever, HollaDieWaldfee, CodingNameKiki, Jeiwan, caventa, KingNFT

Summary

The `openLoan()` function wrongly use `isoUSDLoaned` to calculate `totalUSDborrowed`. Attacker can exploit it to bypass security check and loan isoUSD with no enough collateral.

Vulnerability Detail

vulnerability point

```
function openLoan(
    // ...
    ) external override whenNotPaused
    {
    //...
    uint256 colInUSD = priceCollateralToUSD(currencyKey, _colAmount
                        + collateralPosted[_collateralAddress][msg.sender]);
    uint256 totalUSDborrowed = _USDborrowed
        +  (isoUSDLoaned[_collateralAddress][msg.sender] *
↪   virtualPrice)/LOAN_SCALE;
        // @audit should be isoUSDLoanAndInterest[_collateralAddress][msg.sender]
    require(totalUSDborrowed >= ONE_HUNDRED_DOLLARS, "Loan Requested too small");
    uint256 borrowMargin = (totalUSDborrowed * minOpeningMargin) / LOAN_SCALE;
    require(colInUSD >= borrowMargin, "Minimum margin not met!");

    // ...
}
```

Attack example: <1>Attacker normally loans and produces 10000 isoUSD interest <2>Attacker repays principle but left interest <3>Attacker open a new 10000 isoUSD loan without providing collateral

Impact

Attacker can loan isoUSD with no enough collateral.

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Synths.sol#L120

Tool used

Manual Review

Recommendation

See Vulnerability Detail

Discussion

kree-dotcom

Sponsor confirmed, duplicate of issue #68

kree-dotcom

Fixed https://github.com/kree-dotcom/isomorph/commit/3d77c9f706a52eb6312abc711a007ea8431f749b

Issue H-3: Users who deposit Lyra LP as collateral will lose OP vault rewards

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/78

Found by

0x52

Summary

Optimism currently offers yield farming opportunities for Lyra LPs, see OP Reward Announcement. Every 2 weeks Lyra LPs split a claimable pool. When they use their Lyra LP as collateral, it is transferred to the Lyra vault which means that all OP will be instead claimable by the vault. The vault currently doesn't implement any method to claim or distribute those tokens. The result is a loss of user yield on their Lyra tokens. Aside from the loss of funds, it also highly disincentivizes users from using Lyra tokens as collateral.

Vulnerability Detail

See summary.

Impact

Lyra LPs that use their tokens as collateral will lose all their OP rewards

Code Snippet

Vault_Lyra.sol

Tool used

Manual Review

Recommendation

It's unclear how long OP rewards will continue but it seems like other protocols have been getting 6 months of incentives. Since they are temporary I would recommend not integrating reward distribution directly into the contract. I would recommend adding a function to claim rewards to the Isomorph treasury. After the rewards end, the Isomorph should create an airdrop to distribute those tokens to users during that period of time.

Discussion

kree-dotcom

Sponsor confirmed, will fix

Issue H-4: Malicious user can DOS pool and avoid liquidation by creating secondary liquidity pool for Velodrome token pair

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/72

Found by

0x52

Summary

For every Vault_Velo interaction the vault attempts to price the liquidity of the user. This calls priceLiquidity in the corresponding DepsoitReciept. The prices the underlying assets by swapping them through the Velodrome router. Velodrome can have both a stable and volatile pool for each asset pair. When calling the router directly it routes through the pool that gives the best price. In priceLiquidity the transaction will revert if the router routes through the wrong pool (i.e. trading the volatile pool instead of the stable pool). A malicious user can use this to their advantage to avoid being liquidated. They could manipulate the price of the opposite pool so that any call to liquidate them would route through the wrong pool and revert.

Vulnerability Detail

```
uint256 amountOut; //amount received by trade
bool stablePool; //if the traded pool is stable or volatile.
(amountOut, stablePool) = router.getAmountOut(HUNDRED_TOKENS, token1, USDC);
require(stablePool == stable, "pricing occuring through wrong pool" );
```

DepositReceipt uses the getAmountOut call the estimate the amountOut. The router will return the best rate between the volatile and stable pool. If the wrong pool give the better rate then the transaction will revert. Since pricing is called during liquidation, a malicious user could manipulate the price of the wrong pool so that it returns the better rate and always reverts the liquidation call.

Impact

Malicious user can avoid liquidation

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_USDC.sol#L75-L130

Tool used

Manual Review

Recommendation

Instead of quoting from the router, query the correct pool directly:

```
        uint256 amountOut; //amount received by trade
-       bool stablePool; //if the traded pool is stable or volatile.

-       (amountOut, stablePool) = router.getAmountOut(HUNDRED_TOKENS, token1,
 ↪  USDC);
-       require(stablePool == stable, "pricing occuring through wrong pool" );
+       address pair;

+       pair = router.pairFor(token1, USDC, stable)
+       amountOut = IPair(pair).getAmountOut(HUNDRED_TOKENS, token1)
```

Discussion

kree-dotcom

Sponsor confirmed, will fix.

kree-dotcom

Fixed https://github.com/kree-dotcom/Velo-Deposit-Tokens/commit/58b8f3e14b416630971b7b17b500bbe22d2016aa

Note there are two fixes in this commit relating to the priceLiquidity function. The other fix is for issue #145 , the code for these changes doesn't overlap so should be clear, please ask me if it is not.

Issue H-5: Users are unable close or add to their Lyra vault positions when price is stale or circuit breaker is tripped

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/69

Found by

0x52, rvierdiiev

Summary

Users are unable close or add to their Lyra vault positions when price is stale or circuit breaker is tripped. This is problematic for a few reasons. First is that the circuit breaker can be tripped indefinitely which means their collateral could be frozen forever and they will be accumulating interest the entire time they are frozen. The second is that since they can't add any additional collateral to their loan, the loan may end up being underwater by the time the price is no longer stale or circuit breaker is no longer tripped. They may have wanted to add more assets and now they are liquidated, which is unfair as users who are liquidated are effectively forced to pay a fee to the liquidator.

Vulnerability Detail

```
function _checkIfCollateralIsActive(bytes32 _currencyKey) internal view override
 ↪  {

        //Lyra LP tokens use their associated LiquidityPool to check if they're
        ↪  active
        ILiquidityPoolAvalon LiquidityPool =
        ↪  ILiquidityPoolAvalon(collateralBook.liquidityPoolOf(_currencyKey));
        bool isStale;
        uint circuitBreakerExpiry;
        //ignore first output as this is the token price and not needed yet.
        (, isStale, circuitBreakerExpiry) =
        ↪  LiquidityPool.getTokenPriceWithCheck();
        require( !(isStale), "Global Cache Stale, can't trade");
        require(circuitBreakerExpiry < block.timestamp, "Lyra Circuit Breakers
        ↪  active, can't trade");
}
```

The above lines are run every time a user a user tries to interact with the vault. Currently this is overly restrictive and can lead to a lot of undesired situations, as explained in the summary.

Impact

Frozen assets, unfair interest accumulation and unfair liquidations

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Lyra.sol#L45-L55

Tool used

Manual Review

Recommendation

The contract is frozen when price is stale or circuit breaker is tripped to prevent price manipulation. While it should restrict a majority of actions there are a two that don't need any price validation. If a user wishes to close out their entire loan then there is no need for price validation because the user has no more debt and therefore doesn't need to maintain any level of collateralization. The other situation is if a user adds collateral to their vault and doesn't take out any more loans. In this scenario, the collateralization can only increase, which means that price validation is not necessary.

I recommend the following changes to closeLoan:

```
-   _checkIfCollateralIsActive(currencyKey);
    uint256 isoUSDdebt = (isoUSDLoanAndInterest[_collateralAddress][msg.sender]
    ↪  * virtualPrice) / LOAN_SCALE;
    require( isoUSDdebt >= _USDToVault, "Trying to return more isoUSD than
    ↪  borrowed!");
    uint256 outstandingisoUSD = isoUSDdebt - _USDToVault;
    if(outstandingisoUSD >= TENTH_OF_CENT){ //ignore leftover debts less than
    ↪  $0.001
+       //only need to check collateral value if user has remaining debt
+       _checkIfCollateralIsActive(currencyKey);
        uint256 collateralLeft =
        ↪  collateralPosted[_collateralAddress][msg.sender] - _collateralToUser;
        uint256 colInUSD = priceCollateralToUSD(currencyKey, collateralLeft);
        uint256 borrowMargin = (outstandingisoUSD * minOpeningMargin) /
        ↪  LOAN_SCALE;
        require(colInUSD > borrowMargin , "Remaining debt fails to meet minimum
        ↪  margin!");
    }
```

I recommend removing liquidation threshold check from increaseCollateralAmount:

```
    //debatable check begins here
-   uint256 totalCollat = collateralPosted[_collateralAddress][msg.sender] +
 ↪  _colAmount;
-   uint256 colInUSD = priceCollateralToUSD(currencyKey, totalCollat);
-   uint256 USDborrowed = (isoUSDLoanAndInterest[_collateralAddress][msg.sender]
 ↪  * virtualPrice) / LOAN_SCALE;
-   uint256 borrowMargin = (USDborrowed * liquidatableMargin) / LOAN_SCALE;
-   require(colInUSD >= borrowMargin, "Liquidation margin not met!");
    //debatable check ends here
```

Discussion

kree-dotcom

Sponsor confirmed, will fix. The liquidation threshold for `increaseCollateralAmount()` has been removed as part of the fix for issue #229

Will update when the closeLoan check has been improved.

kree-dotcom

`closeLoan()` fixed here https://github.com/kree-dotcom/isomorph/commit/c14410687e0f2095dd9103b00157b8784430875e

Issue H-6: Outstanding loans cannot be closed or liquidated if collateral is paused

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/57

Found by

0x52, HonorLt

Summary

When a collateral is paused by governance, `collateralValid` is set to false. This causes closing and liquidating of loans to be impossible, leading to two issues. The first is that users with exist loans are unable to close their loans to recover their collateral. The second is that since debt is impossible to liquidate the protocol could end up being stuck with a lot of bad debt.

Vulnerability Detail

```
function pauseCollateralType(
    address _collateralAddress,
    bytes32 _currencyKey
    ) external collateralExists(_collateralAddress) onlyAdmin {
    require(_collateralAddress != address(0)); //this should get caught by the
    ↪  collateralExists check but just to be careful
    //checks two inputs to help prevent input mistakes
    require( _currencyKey == collateralProps[_collateralAddress].currencyKey,
    ↪  "Mismatched data");
    collateralValid[_collateralAddress] = false;
    collateralPaused[_collateralAddress] = true;
}
```

When a collateral is paused `collateralValid[_collateralAddress]` is set to `false`. For `Vault_Lyra` `Vault_Synths` and `Vault_Velo` this will cause `closeLoan` and `callLiquidation` to revert. This traps existing users and prevents liquidations which will result in bad debt for the protocol

Impact

Outstanding loans cannot be closed or liquidated, freezing user funds and causing the protocol to take on bad debt

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/CollateralBook.sol#L185-L195

Tool used

Manual Review

Recommendation

Allow liquidations and loan closure when collateral is paused

Discussion

kree-dotcom

Sponsor confirmed, will fix. We will make it possible to close a loan or liquidate a loan when the collateral is paused.

kree-dotcom

Fixed. https://github.com/kree-dotcom/isomorph/commit/9fef84211c150a6d184b2c492f77fa13b8adc61b

By decoupling the switching of the CollateralValid mapping in the `CollateralBook.sol` from CollateralPaused we can now introduce an additional check in `OpenLoan()` of `require(!collateralBook.collateralPaused(_collateralAddress), "Paused collateral!");` This means `increaseCollateralAmount()`, `closeLoan()` and `callLiquidation()` can all occur for paused collaterals.

Issue H-7: User can steal rewards from other users by withdrawing their Velo Deposit NFTs from other users' depositors

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/51

Found by

0x52, HollaDieWaldfee

Summary

Rewards from staking AMM tokens accumulate to the depositor used to deposit them. The rewards accumulated by a depositor are passed to the owner when they claim. A malicious user to steal the rewards from other users by manipulating other users depositors. Since any NFT of a DepositReceipt can be withdrawn from any depositor with the same DepositReceipt, a malicious user could mint an NFT on their depositor then withdraw in from another user's depositor. The net effect is that that the victims deposits will effectively be in the attackers depositor and they will collect all the rewards.

Vulnerability Detail

```
function withdrawFromGauge(uint256 _NFTId, address[] memory _tokens)  public  {
    uint256 amount = depositReceipt.pooledTokens(_NFTId);
    depositReceipt.burn(_NFTId);
    gauge.getReward(address(this), _tokens);
    gauge.withdraw(amount);
    //AMMToken adheres to ERC20 spec meaning it reverts on failure, no need to
    ↪  check return
    //slither-disable-next-line unchecked-transfer
    AMMToken.transfer(msg.sender, amount);
}
```

Every user must create a `Depositor` using `Templater` to interact with vaults and take loans. `Depositor#withdrawFromGauge` allows any user to withdraw any NFT that was minted by the same `DepositReciept`. This is where the issues arises. Since rewards are accumulated to the `Depositor` in which the underlying is staked a user can deposit to their `Depositor` then withdraw their NFT through the `Depositor` of another user's `Depositor` that uses the same `DepositReciept`. The effect is that the tokens will remained staked to the attackers `Depositor` allowing them to steal all the other user's rewards.

Example: `User A` and `User B` both create a `Depositor` for the same `DepositReciept`.

Both users deposit 100 tokens into their respective `Depositors`. `User B` now calls `withdrawFromGauge` on `Depositor A`. `User B` gets their 100 tokens back and `Depositor B` still has 100 tokens deposited in it. `User B` cannot steal these tokens but they are now collecting the yield on all 100 tokens via `Depositor B` and `User A` isn't getting any rewards at all because `Depositor A` no longer has any tokens deposited into Velodrome gauge.

Impact

Malicious user can steal other user's rewards

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/Depositor.sol#L119-L127

Tool used

Manual Review

Recommendation

Depositors should only be able to burn NFTs that they minted. Change DepositReciept_Base#burn to enforce this:

```
    function burn(uint256 _NFTId) external onlyMinter{
+       //tokens must be burned by the depositor that minted them
+       address depositor = relatedDepositor[_NFTId];
+       require(depositor == msg.sender, "Wrong depositor");
        require(_isApprovedOrOwner(msg.sender, _NFTId), "ERC721: caller is not
        ↪  token owner or approved");
        delete pooledTokens[_NFTId];
        delete relatedDepositor[_NFTId];
        _burn(_NFTId);
    }
```

Discussion

kree-dotcom

Sponsor confirmed, will fix. Appears to be a duplicate of issue #43 's footnote.

kree-dotcom

Fixed https://github.com/kree-dotcom/Velo-Deposit-Tokens/commit/23ff5653b555b11c9f4dead7ff5a72d50eac5788

We have taken a different approach to that suggested by the auditor, theirs looked valid though. We added the checks on lines 82 and 123 that validate any depositReceipt being withdrawn was originally minted by that Depositor contract. Other lines changed in this commit relate to #47 .

Issue H-8: Anyone can withdraw user's Velo Deposit NFT after approval is given to depositor

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/47

Found by

neumo, HonorLt, clems4ever, 0x52, rotcivegaf, bin2chen, HollaDieWaldfee, ak1, CodingNameKiki, Jeiwan

Summary

`Depositor#withdrawFromGauge` is a public function that can be called by anyone which transfers token to `msg.sender`. `withdrawFromGauge` burns the NFT to be withdrawn, which means that `Depositor` must either be approved or be in possession of the NFT. Since it doesn't transfer the NFT to the contract before burning the user must either send the NFT to the `Depositor` or `approve` the `Depositor` in a separate transaction. After the NFT is either transferred or approved, a malicious user could withdraw the NFT for themselves.

Vulnerability Detail

```
function withdrawFromGauge(uint256 _NFTId, address[] memory _tokens)  public  {
    uint256 amount = depositReceipt.pooledTokens(_NFTId);
    depositReceipt.burn(_NFTId);
    gauge.getReward(address(this), _tokens);
    gauge.withdraw(amount);
    //AMMToken adheres to ERC20 spec meaning it reverts on failure, no need to
    ↪  check return
    //slither-disable-next-line unchecked-transfer
    AMMToken.transfer(msg.sender, amount);
}
```

`Depositor#withdrawFromGauge` allows anyone to call it, burning the NFT and sending `msg.sender` the withdrawn tokens.

```
function burn(uint256 _NFTId) external onlyMinter{
    require(_isApprovedOrOwner(msg.sender, _NFTId), "ERC721: caller is not token
    ↪  owner or approved");
    delete pooledTokens[_NFTId];
    delete relatedDepositor[_NFTId];
    _burn(_NFTId);
}
```

`Depositor` calls `DepositReceipt_Base#burn`, which means that it must be either the owner or approved for the NFT. Since `Depositor#withdrawFromGauge` doesn't transfer the NFT from the user, this must happen in a separate transaction. Between the user approval/transfer and them calling `Depositor#withdrawFromGauge` a malicious user could call `Depositor#withdrawFromGauge` first to withdraw the NFT and steal the users funds. This would be very easy to automate with a bot.

Example: `User A` deposits 100 underlying into their `Depositor` and is given `Token A` which represents their deposit. After some time they want to redeem `Token A` so they `Approve` their `Depositor` for `Token A`. `User B` sees the approval and quickly calls `Depositor#withdrawFromGauge` to withdraw `Token A`. `User B` is sent the 100 tokens and `Token A` is burned from `User A`.

Impact

Users attempting to withdraw can have their funds stolen

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/Depositor.sol#L119-L127

Tool used

Manual Review

Recommendation

Only allow owner of NFT to withdraw it:

```
    function withdrawFromGauge(uint256 _NFTId, address[] memory _tokens)  public
    ↪   {
+       require(depositReceipt.ownerOf(_NFTId) == msg.sender);
        uint256 amount = depositReceipt.pooledTokens(_NFTId);
        depositReceipt.burn(_NFTId);
        gauge.getReward(address(this), _tokens);
        gauge.withdraw(amount);
        //AMMToken adheres to ERC20 spec meaning it reverts on failure, no need
        ↪  to check return
        //slither-disable-next-line unchecked-transfer
        AMMToken.transfer(msg.sender, amount);
    }
```

Discussion

kree-dotcom

Sponsor Confirmed, will fix

kree-dotcom

Fixed https://github.com/kree-dotcom/Velo-Deposit-Tokens/commit/23ff5653b555b11c9f4dead7ff5a72d50eac5788

Here we have added a check on line 81 and 122 as suggested. There is also minor refactoring which is needed due to the fact if we are doing a partial withdrawal then after calling `depositReceipt.split()` the owner of the newly acquired depositReceipt is the Depositor not the original msg.sender. Therefore we moved the withdrawal logic to an internal function that both `withdrawFromGauge()` and `partialWithdrawFromGauge()` both access after ownership checks.

Issue H-9: Swapping 100 tokens in DepositReceipt_ETH and DepositReciept_USDC breaks usage of WBTC LP and other high value tokens

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/46

Found by

0x52, clems4ever

Summary

DepositReceipt_ETH and DepositReciept_USDC checks the value of liquidity by swapping 100 tokens through the swap router. WBTC is a good example of a token that will likely never work as LP due to the massive value of swapping 100 WBTC. This makes DepositReceipt_ETH and DepositReciept_USDC revert during slippage checks after calculating amount out. As of the time of writing this, WETH also experiences a 11% slippage when trading 100 tokens. Since DepositReceipt_ETH only supports 18 decimal tokens, WETH/USDC would have to use DepositReciept_USDC, resulting in WETH/USDC being incompatible. The fluctuating liquidity could also make this a big issue as well. If liquidity reduces after deposits are made, user deposits could be permanently trapped.

Vulnerability Detail

```
//check swap value of 100tokens to USDC to protect against flash loan attacks
uint256 amountOut; //amount received by trade
bool stablePool; //if the traded pool is stable or volatile.
(amountOut, stablePool) = router.getAmountOut(HUNDRED_TOKENS, token1, USDC);
```

The above lines try to swap 100 tokens from token1 to USDC. In the case of WBTC 100 tokens is a monstrous amount to swap. Given the low liquidity on the network, it simply won't function due to slippage requirements.

```
function _priceCollateral(IDepositReceipt depositReceipt, uint256 _NFTId)
↪   internal view returns(uint256){
    uint256 pooledTokens = depositReceipt.pooledTokens(_NFTId);
    return( depositReceipt.priceLiquidity(pooledTokens));
}

function totalCollateralValue(address _collateralAddress, address _owner) public
↪   view returns(uint256){
    NFTids memory userNFTs = loanNFTids[_collateralAddress][_owner];
    IDepositReceipt depositReceipt = IDepositReceipt(_collateralAddress);
```

```
    //slither-disable-next-line uninitialized-local-variables
    uint256 totalPooledTokens;
    for(uint256 i =0; i < NFT_LIMIT; i++){
        //check if each slot contains an NFT
        if (userNFTs.ids[i] != 0){
            totalPooledTokens += depositReceipt.pooledTokens(userNFTs.ids[i]);
        }
    }
    return(depositReceipt.priceLiquidity(totalPooledTokens));
}
```

One of the two functions above are used to price LP for every vault action on Vault_Velo. If liquidity is sufficient when user deposits but then drys up after, the users deposit would be permanently trapped in the in the vault. In addition to this liquidation would also become impossible causing the protocol to assume bad debt.

This could also be exploited by a malicious user. First they deposit a large amount of collateral into the Velodrome WBTC/USDC pair. They take a portion of their LP and take a loan against it. Now they withdraw the rest of their LP. Since there is no longer enough liquidity to swap 100 tokens with 5% slippage, they are now safe from liquidation, allowing a risk free loan.

Impact

LPs that contain high value tokens will be unusable at best and freeze user funds or be abused at the worst case

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_ETH.sol#L93-L152

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_USDC.sol#L75-L130

Tool used

Manual Review

Recommendation

Change the number of tokens to an immutable, so that it can be set individually for each token. Optionally you can add checks (shown below) to make sure that the number of tokens being swapped will result in at least some minimum value of USDC is received. Similar changes should be made for DepositReceipt_ETH:

```
constructor(string memory _name,
            string memory _symbol,
            address _router,
            address _token0,
            address _token1,
            uint256 _tokensToSwap,
            bool _stable,
            address _priceFeed)
            ERC721(_name, _symbol){

    ...

    if (keccak256(token0Symbol) == keccak256(USDCSymbol)){
        require( IERC20Metadata(_token1).decimals() == 18, "Token does not have
        ↪  18dp");

+       (amountOut,) = _router.getAmountOut(_tokensToSwap, token1, USDC);

+       //swapping tokens must yield at least 100 USDC
+       require( amountOut >= 1e8);
+       tokensToSwap = _tokensToSwap;
    }
    else
    {
        bytes memory token1Symbol =
        ↪  abi.encodePacked(IERC20Metadata(_token1).symbol());
        require( keccak256(token1Symbol) == keccak256(USDCSymbol), "One token
        ↪  must be USDC");
        require( IERC20Metadata(_token0).decimals() == 18, "Token does not have
        ↪  18dp");

+       (amountOut, ) = _router.getAmountOut(_tokensToSwap, token0, USDC);

+       //swapping tokens must yield at least 100 USDC
+       require( amountOut >= 1e8);
+       tokensToSwap = _tokensToSwap;
    }
```

Discussion

kree-dotcom

Sponsor confirmed, will fix

kree-dotcom

Fixed https://github.com/kree-dotcom/Velo-Deposit-Tokens/commit/76bb63d885

759825d93e95925364806168d02e51

Both DepositReceipts have had the swap quantity changed to an immutable. DepositReceipt_USDC contains further checks to ensure the swap quantity would receive between 100 and 105 USDC on deployment so we know it is the right scale. This check could also be done with DepositReceipt_ETH but because the value of ETH is dynamic it would be a little more complex so instead we just included a non-zero value check.

Sorry, corrected a mistake with still using the `HUNDRED` constant for scaling this trade https://github.com/kree-dotcom/Velo-Deposit-Tokens/commit/d8e7651f44a4cee07d23c00aa6ee612e25879771

Issue M-1: DepositReceipt_Base.sol#L21 : HEARTBEAT_-TIME gap is too huge

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/256

Found by

ak1

Summary

HEARTBEAT_TIME = 24 hours could not be safe. The oracle data still be stale one.

Vulnerability Detail

oracle is using the HEARTBEAT_TIME as 24 hours. Since the price of oracle could vary in the time gap of 3 hours, using 24 hours could be still dangerous.

Impact

Stale data used. Front runnable issue.

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_Base.sol#L21

Tool used

Manual Review

Recommendation

Use 3 hours as hearbeat.

Discussion

kree-dotcom

Sponsor Confirmed. 24Hrs was originally for the sUSD chainlink feed, as seen here it can and should be much lower for other price feeds https://data.chain.link/optimism/mainnet/crypto-usd/eth-usd (mouse over trigger parameters)

kree-dotcom

clicked wrong button

kree-dotcom

3 hours as recommended by the auditor does not seem sufficient. Some Optimism price feeds such as ETH/USD and OP/USD have Heartbeats of 1200s or 20min. Currently we cannot find a method to fetch this via the oracle address and it looks like the Heartbeat sensitivity would have to be set per deployment of the depositReceipt.

kree-dotcom

Fixed, https://github.com/kree-dotcom/Velo-Deposit-Tokens/commit/398f40cce538461396966ac22273f846d56f6f27

`HEARTBEAT_TIME` is now an immutable var that is set by the deployer for each instance of a depositReceipt.

Issue M-2: Lyra vault underestimates the collateral value

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/242

Found by

hansfriese

Summary

Lyra vault subtracts the withdrawal fee while calculating the collateral value in USD, and it does not match the actual Lyra Pool implementation.

Vulnerability Detail

The user's collateral value is estimated using the function `priceCollateralToUSD()` at `Vault_Lyra.sol#L77` as follows.

```
function priceCollateralToUSD(bytes32 _currencyKey, uint256 _amount) public view
↪   override returns(uint256){
        //The LiquidityPool associated with the LP Token is used for pricing
    ILiquidityPoolAvalon LiquidityPool =
↪   ILiquidityPoolAvalon(collateralBook.liquidityPoolOf(_currencyKey));
    //we have already checked for stale greeks so here we call the basic price
↪   function.
    uint256 tokenPrice = LiquidityPool.getTokenPrice();
    uint256 withdrawalFee = _getWithdrawalFee(LiquidityPool);
    uint256 USDValue  = (_amount * tokenPrice) / LOAN_SCALE;
    //we remove the Liquidity Pool withdrawalFee
    //as there's no way to remove the LP position without paying this.
    uint256 USDValueAfterFee = USDValue * (LOAN_SCALE- withdrawalFee)/LOAN_SCALE;
    return(USDValueAfterFee);
}
```

So it is understood that the withdrawal fee is removed to get the reasonable value of the collateral. But according to the Lyra Pool implementation, the token price used for withdrawal is calculated using the function `_getTotalBurnableTokens`. And the function `_getTotalBurnableTokens` is as belows.

```
function _getTotalBurnableTokens()
    internal
    returns (
      uint tokensBurnable,
      uint tokenPriceWithFee,
      bool stale
    )
```

```
  {
    uint burnableLiquidity;
    uint tokenPrice;
    (tokenPrice, stale, burnableLiquidity) = _getTokenPriceAndStale();

    if (optionMarket.getNumLiveBoards() != 0) {
      tokenPriceWithFee = tokenPrice.multiplyDecimal(DecimalMath.UNIT -
↪   lpParams.withdrawalFee);
    } else {
      tokenPriceWithFee = tokenPrice;//@audit withdrawalFee is not applied if
↪   there are no live borads
    }

    return (burnableLiquidity.divideDecimal(tokenPriceWithFee),
↪   tokenPriceWithFee, stale);
  }
```

From the code, it is clear that the withdrawal fee is subtracted only when the related option market has live boards. Because `Vault_Lyra.sol` applies a withdrawal fee all the time to price the collateral, it means the user's collateral is under-valued.

Impact

User's collaterals are under-valued than reasonable and might get to a liquidatable status sooner than expected. A liquidator can abuse this to get an unfair profit by liquidating the user's collateral with the under-estimated value and withdrawing it from the Lyra pool without paying a withdrawal fee.

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Lyra.sol#L77

Tool used

Manual Review

Recommendation

Make sure to apply withdrawal fee consistent to how Lyra pool does.

Discussion

kree-dotcom

Sponsor confirmed, will fix

kree-dotcom

Currently this fix looks like it will be quite a lot of alterations. If there is time and it doesn't introduce too many changes for Sherlock to check we will try to fix it.

The reason is the call added must be made to the optionMarket contract of each collateral, this address is stored as an internal address in liquidityPool and other contracts, therefore to access it we must add another field to Lyra collateral in the collateral book, this would involve altering all collateralBook functions to expect another field as well as other vaults and systems to adhere to this larger model.

Alternatively we can add another mapping to the Vault_Lyra to store the OptionsMarkets and an admin only function to add new ones but this feels like poor design.

Leaving this issue unfixed is unlikely to cause large problems. LiquidityTokens should only have zero live boards if an optionMarket is closed (usually to be depreciated) with zero live boards the collateral should be earning no interest for the owner and therefore they would likely desire to close their loan and redeem their collateral and so not be in a situation to be liquidated.

Issue M-3: The protocol shouldn't charge interests when paused

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/234

Found by

hansfriese, rvierdiiev

Summary

The protocol charges interest from users using `virtualPrice` and it increases when the protocol is paused.

As a result, users would be forced to pay more interests and experience an unexpected liquidation.

Vulnerability Detail

The protocol has 3 kinds of the vault and each one has `pause/unpause` option by `pausers`.

Also, each collateral would be paused using `CollateralBook.pauseCollateralType()`.

But it updates the `virtualPrice` during the paused period and the below scenarios would be possible.

Scenario 1

1. A user `Alice` opened a loan using some collaterals.
2. The vault was paused for a while for some unexpected reason.
3. Meanwhile, her loan was changed to a `liquidatable` one but she can't add collaterals(or close the loan) in the paused state.
4. After the protocol is unpaused, she's trying to protect her loan by adding collaterals but `Bob` can liquidate her loan with front running.
5. Even if her loan isn't liquidated, she should pay interests during the paused period and it's not fair for her.

Scenario 2

1. A user `Alice` opened a loan with `minOpeningMargin = 101%`.
2. After the protocol was paused for some reason, the admin decided to change `minOpeningMargin = 105%`.

3. `Alice` wants to close her loan before it's applied because it's too high for her but she can't because it's paused.
4. After the new `minOpeningMargin` is applied, `Alice` will be forced to pay interests of the higher `minOpeningMargin` for the paused period.

When I check other protocols to charge interests, it's normal to enable some ways to protect their loans during the paused period for users.

Currently, all functions don't work in the paused mode and it shouldn't charge interests in this case.

Impact

Users might be forced to pay more interests or their loans might be liquidated unexpectedly.

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/CollateralBook.sol#L127 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Base_ERC20.sol#L85 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Velo.sol#L141 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Base_ERC20.sol#L203-L221 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Velo.sol#L248-L265

Tool used

Manual Review

Recommendation

We shouldn't increase the `virtualPrice` during the paused period.

Discussion

kree-dotcom

Sponsor confirmed, will fix. Partial duplicate of issues #42 and issue #38. This was a design decision but we have decided to change it so that users can closeLoans, addCollateral or be liquidated when a collateral/vault is Paused.

kree-dotcom

Allow users to call `increaseCollateralAmount()`, `closeLoan()` and `callLiquidation()` of each Vault while the vault is paused. https://github.com/kree-dotcom/isomorph/commit/627212dcdcc3c22553de5587a90c9fae211a4888

kree-dotcom

Freeze collateral interest when collateral is paused. https://github.com/kree-dotcom/isomorph/commit/8ed4909462315bda79a08b773c530dfadfc1c4a3

This is achieved by ensuring the virtualPrice is up-to-date when pausing and then altering the lastUpdateTime when unpausing so that the system thinks the virtualPrice has been updated for the paused time period.

If a collateral is paused and unpaused often then interest owed can be lost because of rounding here as we can lose <180s of interest due per pausing. However as only admins can pause collaterals this is deemed an insignificant risk to the system as it would just slightly reduce fees owed on one collateral.

Issue M-4: Wrong constants for time delay

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/231

Found by

neumo, 0x4non, GimelSec, rvierdiiev, hansfriese, wagmi, jonatascm

Summary

This protocol uses several constants for time dealy and some of them are incorrect.

Vulnerability Detail

In `isoUSDToken.sol`, `ISOUSD_TIME_DELAY` should be `3 days` instead of 3 seconds.

```
uint256 constant ISOUSD_TIME_DELAY = 3; // days;
```

In `CollateralBook.sol`, `CHANGE_COLLATERAL_DELAY` should be `2 days` instead of 200 seconds.

```
uint256 public constant CHANGE_COLLATERAL_DELAY = 200; //2 days
```

Impact

Admin settings would be updated within a short period of delay so that users wouldn't react properly.

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/isoUSDToken.sol#L10 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/CollateralBook.sol#L23

Tool used

Manual Review

Recommendation

2 constants should be modified as mentioned above.

Discussion

kree-dotcom

Sponsor confirmed, will fix.

kree-dotcom

Fixed https://github.com/kree-dotcom/isomorph/commit/4fc80e6178204691a365f656908c278d5faf4f88 , woops then forgot a semicolon, this was added here https://github.com/kree-dotcom/isomorph/commit/9bad2748dd3f3e7905dc8013383aef0cf98b1bea

isoToken was not altered in this commit but is correct. I made a copying error when setting up the Audit repo originally.

https://github.com/kree-dotcom/isomorph/blob/4fc80e6178204691a365f656908c278d5faf4f88/contracts/isoUSDToken.sol#L10

Issue M-5: increaseCollateralAmount : User is not allowed to increase collateral freely.

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/229

Found by

rvierdiiev, ak1, bin2chen

Summary

For all the tree type of vault, a user is allowed to increase collateral only if the overall collateral value is higher than the margin value.

imo, this restriction may not be needed. anyway user is adding the collateral that could eventually save from liquidation.

Protocol will loose advantage due to this restriction.

Vulnerability Detail

Codes from lyra vault implementation :

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Lyra.sol#L155-L192

Line 184

```
require(colInUSD >= borrowMargin, "Liquidation margin not met!");
```

For synth - Refer here

For velo - Refer here

Impact

User may not have the collateral all at once, but they can add like an EMI.

Protocol will loose the repayment anyway.

What is no one comes for liquidation - again this could lose.

Code Snippet

Refer vulnerability section

Tool used

Manual Review

Recommendation

Allow user add collateral freely.

Discussion

kree-dotcom

Sponsor confirmed, will fix. Issue #41 is a duplicate of this

kree-dotcom

Fixed https://github.com/kree-dotcom/isomorph/commit/6b403bad09388fca0153b843a2e552b5e3d235cd

Issue M-6: Dangerous assumption on the peg of USDC can lead to manipulations

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/224

Found by

Deivitto, yixxas, Jeiwan

Summary

Dangerous assumption on the peg of USDC can lead to manipulations

Vulnerability Detail

When pricing liquidity of a Velodrome USDC pool, it's assumed that hte price of USDC is exactly $1 (DepositReceipt_USDC.sol#L100, DepositReceipt_USDC.sol#L123). However, in reality, there's no hard peg, the price can go both above or below $1 (https://coinmarketcap.com/currencies/usd-coin/).

The volatility of USDC will also affect the price of the other token in the pool since it's priced in USDC (DepositReceipt_USDC.sol#L87, DepositReceipt_USDC.sol#L110) and then compared to its USD price from a Chainlink oracle (DepositReceipt_USDC.sol#L90-L98).

This issue is also applicable to the hard coded peg of sUSD when evaluating the USD price of a Synthetix collateral (Vault_Synths.sol#L76):

```
/// @return returns the value of the given synth in sUSD which is assumed to be
 ↪  pegged at $1.
function priceCollateralToUSD(bytes32 _currencyKey, uint256 _amount) public view
 ↪  override returns(uint256){
    //As it is a synth use synthetix for pricing
    return (synthetixExchangeRates.effectiveValue(_currencyKey, _amount,
 ↪  SUSD_CODE));
}
```

And sUSD is even less stable than USDC (https://coinmarketcap.com/currencies/susd/).

Together with isoUSD not having a stability mechanism, these assumptions can lead to different manipulations with the price of isoUSD and the arbitraging opportunities created by the hard peg assumptions (sUSD and USDC will be priced differently on exchanges and on Isomorph).

Impact

If the price of USDC falls below $1, collateral will be priced higher than expected. This will keep borrowers from being liquidated. And it will probably affect the price of isoUSD since there will be an arbitrage opportunity: the cheaper USDC will be priced higher as collateral on Isomorph. If hte price of USDC raises above $1, borrowers' collateral will be undervalued and some liquidations will be possible that wouldn't have be allowed if the actual price of USDC was used.

Code Snippet

The value of USDC equals its amount (DepositReceipt_USDC.sol#L100, DepositReceipt_USDC.sol#L123):

```
value0 = token0Amount * SCALE_SHIFT;
```

The other token in a pool is priced in USDC (DepositReceipt_USDC.sol#L87, DepositReceipt_USDC.sol#L110):

```
(amountOut, stablePool) = router.getAmountOut(HUNDRED_TOKENS, token1, USDC);
```

Tool used

Manual Review

Recommendation

Consider using the Chainlink USDC/USD feed to get the price of USDC and price liquidity using the actual price of USDC. Also, consider converting sUSD prices of Synthetix collaterals to USD to mitigate the discrepancy in prices between external exchanges and Isomorph.

Discussion

kree-dotcom

Sponsor confirmed however this is part of the protocol design. That said now that we have the duel oracle system for DepositReceipt_ETH it should not be too difficult to replicate this with minor changes so that the USDC value in DepositReceipt_USDC uses a chainlink oracle also.

As for sUSD, we will explore how changes might impact the system. The system is already designed to absorb small fluctuations in the value of sUSD by having a gap between the opening margin and the liquidation margin. However we can see that it can be unfair to a user if they get liquidated because they fairly price their Synth collateral using sUSD = $1.01 rather than the hardcoded exchange rate.

Issue M-7: `latestRoundData()` has no check for round completeness

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/200

Found by

HonorLt, yixxas, __141345__, 8olidity, caventa

Summary

No check for round completeness could lead to stale prices and wrong price return value, or outdated price. The functions rely on accurate price feed might not work as expected, sometimes can lead to fund loss.

Vulnerability Detail

The oracle wrapper `getOraclePrice()` call out to an oracle with `latestRoundData()` to get the price of some token. Although the returned timestamp is checked, there is no check for round completeness.

According to Chainlink's documentation, this function does not error if no answer has been reached but returns 0 or outdated round data. The external Chainlink oracle, which provides index price information to the system, introduces risk inherent to any dependency on third-party data sources. For example, the oracle could fall behind or otherwise fail to be maintained, resulting in outdated data being fed to the index price calculations. Oracle reliance has historically resulted in crippled on-chain systems, and complications that lead to these outcomes can arise from things as simple as network congestion.

Reference

Chainlink documentation:
https://docs.chain.link/docs/historical-price-data/#historical-rounds

Impact

If there is a problem with chainlink starting a new round and finding consensus on the new value for the oracle (e.g. chainlink nodes abandon the oracle, chain congestion, vulnerability/attacks on the chainlink system) consumers of this contract may continue using outdated stale data (if oracles are unable to submit no new round is started).

This could lead to stale prices and wrong price return value, or outdated price.

As a result, the functions rely on accurate price feed might not work as expected, sometimes can lead to fund loss. The impacts vary and depends on the specific situation like the following:

- incorrect liquidation
  - some users could be liquidated when they should not
  - no liquidation is performed when there should be
- wrong price feed
  - causing inappropriate loan being taken, beyond the current collateral factor
  - too low price feed affect normal bor

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_Base.sol#L164-L181

Tool used

Manual Review

Recommendation

Validate data feed for round completeness:

```
function getOraclePrice(IAggregatorV3 _priceFeed, int192 _maxPrice, int192
↪  _minPrice) public view returns (uint256 ) {
    (
        uint80 roundID,
        int signedPrice,
        /*uint startedAt*/,
        uint timeStamp,
        uint80 answeredInRound
    ) = _priceFeed.latestRoundData();
    //check for Chainlink oracle deviancies, force a revert if any are present.
↪  Helps prevent a LUNA like issue
    require(signedPrice > 0, "Negative Oracle Price");
    require(timeStamp >= block.timestamp - HEARTBEAT_TIME , "Stale pricefeed");
    require(signedPrice < _maxPrice, "Upper price bound breached");
    require(signedPrice > _minPrice, "Lower price bound breached");
    require(answeredInRound >= roundID, "round not complete");

    uint256 price = uint256(signedPrice);
```

```
    return price;
}
```

Discussion

kree-dotcom

Sponsor confirmed, will fix.

kree-dotcom

Fixed https://github.com/kree-dotcom/Velo-Deposit-Tokens/commit/5c656e791e70ecdfe26f9807438498384d7f6108

Issue M-8: Wrong `CHANGE_COLLATERAL_DELAY` in Collateral-Book

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/191

Found by

yixxas, GimelSec, rvierdiiev, ctf_sec, CodingNameKiki, Jeiwan, 0xjayne

Summary

Admins can bypass time delay due to the wrong value of `CHANGE_COLLATERAL_DELAY`.

Vulnerability Detail

The comment shows that the `CHANGE_COLLATERAL_DELAY` should be 2 days, but it's only 200 which means 3 minutes and 20 seconds.

Impact

Admin can bypass the 2 days time delay and only need to wait less than 5 minutes to call `changeCollateralType`.

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/CollateralBook.sol#L23 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/CollateralBook.sol#L130

Tool used

Manual Review

Recommendation

```
uint256 public constant CHANGE_COLLATERAL_DELAY = 2 days; //2 days
```

Discussion

kree-dotcom

Sponsor confirmed, will fix. Duplicate of issue #231

Issue M-9: Bad debt may persist even after complete liquidation in Velo Vault due to truncation

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/174

Found by

0x52

Summary

When liquidating a user, if all their collateral is taken but it is not valuable enough to repay the entire loan they would be left with remaining debt. This is what is known as bad debt because there is no collateral left to take and the user has no obligation to pay it back. When this occurs, the vault will forgive the user's debts, clearing the bad debt. The problem is that the valuations are calculated in two different ways which can lead to truncation issue that completely liquidates a user but doesn't clear their bad debt.

Vulnerability Detail

```
uint256 totalUserCollateral = totalCollateralValue(_collateralAddress,
↪  _loanHolder);
uint256 proposedLiquidationAmount;
{ //scope block for liquidationAmount due to stack too deep
    uint256 liquidationAmount = viewLiquidatableAmount(totalUserCollateral, 1
    ↪  ether, isoUSDBorrowed, liquidatableMargin);
    require(liquidationAmount > 0 , "Loan not liquidatable");
    proposedLiquidationAmount =
    ↪  _calculateProposedReturnedCapital(_collateralAddress, _loanNFTs,
    ↪  _partialPercentage);
    require(proposedLiquidationAmount <= liquidationAmount, "excessive
    ↪  liquidation suggested");
}
uint256 isoUSDreturning =
↪  proposedLiquidationAmount*LIQUIDATION_RETURN/LOAN_SCALE;
if(proposedLiquidationAmount >= totalUserCollateral){
    //@audit bad debt cleared here
}
```

The primary check before clearing bad debt is to check if `proposedLiquidationAmount >= totalUserCollateral`. The purpose of this check is to confirm that all of the user's collateral is being liquidated. The issue is that each value is calculated differently.

```
function totalCollateralValue(address _collateralAddress, address _owner) public
↪  view returns(uint256){
    NFTids memory userNFTs = loanNFTids[_collateralAddress][_owner];
    IDepositReceipt depositReceipt = IDepositReceipt(_collateralAddress);
    //slither-disable-next-line uninitialized-local-variables
    uint256 totalPooledTokens;
    for(uint256 i =0; i < NFT_LIMIT; i++){
        //check if each slot contains an NFT
        if (userNFTs.ids[i] != 0){
            totalPooledTokens += depositReceipt.pooledTokens(userNFTs.ids[i]);
        }
    }
    return(depositReceipt.priceLiquidity(totalPooledTokens));
}
```

`totalCollateralValue` it used to calculate `totalUserCollateral`. In this method the pooled tokens are summed across all NFT's then they are priced. This means that the value of the liquidity is truncated exactly once.

```
function _calculateProposedReturnedCapital(
    address _collateralAddress,
    CollateralNFTs calldata _loanNFTs,
    uint256 _partialPercentage
    ) internal view returns(uint256){
    //slither-disable-next-line uninitialized-local-variables
    uint256 proposedLiquidationAmount;
    require(_partialPercentage <= LOAN_SCALE, "partialPercentage greater than
    ↪  100%");
    for(uint256 i = 0; i < NFT_LIMIT; i++){
            if(_loanNFTs.slots[i] < NFT_LIMIT){
                if((i == NFT_LIMIT -1) && (_partialPercentage > 0) &&
                ↪  (_partialPercentage < LOAN_SCALE) ){
                    //final slot is NFT that will be split if necessary
                    proposedLiquidationAmount +=
                                                (( _priceCollateral(IDepositRece
                                                ↪  ipt(_collateralAddress),
                                                ↪  _loanNFTs.ids[i])
                                                *_partialPercentage)/
                                                ↪  LOAN_SCALE);
                }
                else {
                    proposedLiquidationAmount +=
                    ↪  _priceCollateral(IDepositReceipt(_collateralAddress),
                    ↪  _loanNFTs.ids[i]);
                }
            }
```

```
        }
    return proposedLiquidationAmount;
}
```

`_calculateProposedReturnedCapital` is used to calculate `proposedLiquidationAmount`. The key difference is that each NFT is priced individually. The result is that the value is truncated up to NFT_LIMIT times. This can lead to proposedLiquidationAmount being less than totalUserCollateral even if all user collateral is being liquidated.

Example: User A has 2 NFTs. They are valued as follows assuming no truncation: 10.6 and 10.7. When calculating via `totalCollateralValue` they will be summed before they are truncated while in `_calculateProposedReturnedCapital` they will be truncated before they are summed.

totalCollateralValue: 10.6 + 10.7 = 21.3 => 21 (truncated)

_calculateProposedReturnedCapital: 10.6 => 10 (truncated) 10.7 => 10 (truncated)

10 + 10 = 20

As shown above when using the exact same inputs into our two different functions the final answer is different. In a scenario like this, even though all collateral is taken from the user, their bad debt won't be cleared.

Impact

Bad debt will not be cleared in some liquidation scenarios

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Velo.sol#L593-L619

Tool used

Manual Review

Recommendation

`_calculateProposedReturnedCapital` should be changed to be similar to `totalCollateralValue`, summing all pooled tokens before pricing:

```
    function _calculateProposedReturnedCapital(
        address _collateralAddress,
        CollateralNFTs calldata _loanNFTs,
        uint256 _partialPercentage
    ) internal view returns(uint256) {
```

```
+       IDepositReceipt depositReceipt = IDepositReceipt(_collateralAddress);
        //slither-disable-next-line uninitialized-local-variables
+       uint256 totalPooledTokens
-       uint256 proposedLiquidationAmount;
        require(_partialPercentage <= LOAN_SCALE, "partialPercentage greater
        ↪  than 100%");
        for(uint256 i = 0; i < NFT_LIMIT; i++){
            if(_loanNFTs.slots[i] < NFT_LIMIT){
                if((i == NFT_LIMIT -1) && (_partialPercentage > 0) &&
                ↪  (_partialPercentage < LOAN_SCALE) ){
                    //final slot is NFT that will be split if necessary
+                   totalPooledTokens +=
↪   ((depositReceipt.pooledTokens(userNFTs.ids[i]) * _partialPercentage) /
↪   LOAN_SCALE);
-                   proposedLiquidationAmount +=
-                                                ((
↪   _priceCollateral(IDepositReceipt(_collateralAddress), _loanNFTs.ids[i])
-                                               *_partialPercentage)/ LOAN_SCALE);
                }
                else{
+                   totalPooledTokens +=
↪   depositReceipt.pooledTokens(userNFTs.ids[i]);
-                   proposedLiquidationAmount +=
↪   _priceCollateral(IDepositReceipt(_collateralAddress), _loanNFTs.ids[i]);
                }
            }
        }
+       return(depositReceipt.priceLiquidity(totalPooledTokens));
-       return proposedLiquidationAmount;
    }
```

Discussion

kree-dotcom

Sponsor confirmed, fixed https://github.com/kree-dotcom/isomorph/commit/6c0bd26136ff4b33f23437551e22b1066156995b

Sorry there were some errors I've corrected in the above commit https://github.com/kree-dotcom/isomorph/commit/4322e27fa3e61f1515ba8e88d275676c7b9ed915

Issue M-10: Vault_Base_ERC20#_updateVirtualPrice calculates interest incorrectly if updated frequently

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/158

Found by

libratus, 0x52, Atarpara, bin2chen, rvierdiiev, HollaDieWaldfee, wagmi, Jeiwan, KingNFT, hansfriese

Summary

Updating the virtual price of an asset happens in discrete increments of 3 minutes. This is done to reduce the chance of DOS loops. The issue is that it updates the time to an incorrect timestamp. It should update to the truncated 3 minute interval but instead updates to the current timestamp. The result is that the interest calculation can be abused to lower effective interest rate.

Vulnerability Detail

```
function _updateVirtualPrice(uint256 _currentBlockTime, address
↪  _collateralAddress) internal {
    (   ,
        ,
        ,
        uint256 interestPer3Min,
        uint256 lastUpdateTime,
        uint256 virtualPrice,

    ) = _getCollateral(_collateralAddress);
    uint256 timeDelta = _currentBlockTime - lastUpdateTime;
    //exit gracefully if two users call the function for the same collateral in
    ↪  the same 3min period
    //@audit increments
    uint256 threeMinuteDelta = timeDelta / 180;
    if(threeMinuteDelta > 0) {
        for (uint256 i = 0; i < threeMinuteDelta; i++ ){
        virtualPrice = (virtualPrice * interestPer3Min) / LOAN_SCALE;
        }
        collateralBook.vaultUpdateVirtualPriceAndTime(_collateralAddress,
        ↪  virtualPrice, _currentBlockTime);
    }
}
```

_updateVirtualPrice is used to update the interest calculations for the specified collateral and is always called with block.timestamp. Due to truncation threeMinuteDelta is always rounded down, that is if there has been 1.99 3-minute intervals it will truncate to 1. The issue is that in the collateralBook#vaultUpdateVirtualPriceAndTime subcall the time is updated to block.timestamp (_currentBlockTime).

Example: lastUpdateTime = 1000 and block.timestamp (_currentBlockTime) = 1359.

timeDelta = 1359 - 1000 = 359

threeMinuteDelta = 359 / 180 = 1

This updates the interest by only as single increment but pushes the new time forward 359 seconds. When called again it will use 1359 as lastUpdateTime which means that 179 seconds worth of interest have been permanently lost. Users with large loan positions could abuse this to effectively halve their interest accumulation. Given how cheap optimism transactions are it is highly likely this could be exploited profitably with a bot.

Impact

Interest calculations will be incorrect if they are updated frequently, which can be abused by users with large amounts of debt to halve their accumulated interest

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Base_ERC20.sol#L203-L221

Tool used

Manual Review

Recommendation

Before updating the interest time it should first truncate it to the closest 3-minute interval:

```
    if(threeMinuteDelta > 0) {
        for (uint256 i = 0; i < threeMinuteDelta; i++ ){
            virtualPrice = (virtualPrice * interestPer3Min) / LOAN_SCALE;
        }
+       _currentBlockTime = (_currentBlockTime / 180) * 180;
        collateralBook.vaultUpdateVirtualPriceAndTime(_collateralAddress,
        ↪  virtualPrice, _currentBlockTime);
    }
```

Discussion

kree-dotcom

Sponsor confirmed, will fix.

kree-dotcom

Fixed https://github.com/kree-dotcom/isomorph/commit/ae410496c024af5b061cf85997f225ae46fd56e6

Issue M-11: priceLiquidity() may not work if PriceFeed.aggregator is updated

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/145

Found by

cccz

Summary

priceLiquidity() may not work if PriceFeed.aggregator() is updated

Vulnerability Detail

In the constructor of the DepositReceipt_* contract, the value of minAnswer/maxAnswer in priceFeed.aggregator() is obtained and assigned to *MinPrice/*MaxPrice as the maximum/minimum price limit when calling the getOraclePrice function in priceLiquidity, and *MinPrice/*MaxPrice can not change.

```
        IAccessControlledOffchainAggregator  aggregator =
↪   IAccessControlledOffchainAggregator(priceFeed.aggregator());
        //fetch the pricefeeds hard limits so we can be aware if these have been
↪   reached.
        tokenMinPrice = aggregator.minAnswer();
        tokenMaxPrice = aggregator.maxAnswer();
...
            uint256 oraclePrice = getOraclePrice(priceFeed, tokenMaxPrice,
↪   tokenMinPrice);
...
    function getOraclePrice(IAggregatorV3 _priceFeed, int192 _maxPrice, int192
↪   _minPrice) public view returns (uint256 ) {
        (
            /*uint80 roundID*/,
            int signedPrice,
            /*uint startedAt*/,
            uint timeStamp,
            /*uint80 answeredInRound*/
        ) = _priceFeed.latestRoundData();
        //check for Chainlink oracle deviancies, force a revert if any are
↪   present. Helps prevent a LUNA like issue
        require(signedPrice > 0, "Negative Oracle Price");
        require(timeStamp >= block.timestamp - HEARTBEAT_TIME , "Stale
↪   pricefeed");
        require(signedPrice < _maxPrice, "Upper price bound breached");
```

```
        require(signedPrice > _minPrice, "Lower price bound breached");
```

But in the priceFeed contract, the address of the aggregator can be changed by the owner, which may cause the value of minAnswer/maxAnswer to change, and the price limit in the DepositReceipt_* contract to be invalid, and priceLiquidity() can not work.

```
function confirmAggregator(address _aggregator)
  external
  onlyOwner()
{
  require(_aggregator == address(proposedAggregator), "Invalid proposed
↪  aggregator");
  delete proposedAggregator;
  setAggregator(_aggregator);
}



/*
 * Internal
 */

function setAggregator(address _aggregator)
  internal
{
  uint16 id = currentPhase.id + 1;
  currentPhase = Phase(id, AggregatorV2V3Interface(_aggregator));
  phaseAggregators[id] = AggregatorV2V3Interface(_aggregator);
}
...
  function aggregator()
  external
  view
  returns (address)
{
  return address(currentPhase.aggregator);
}
```

Impact

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_ETH.sol#L66-L80 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_USDC.sol#L60-L64 https://github.com/sherlock-audit/2022-11-iso

morph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_ETH.sol#L107-L109 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_ETH.sol#L134-L135 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_USDC.sol#L90-L91 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_USDC.sol#L113-L114 https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Velo-Deposit-Tokens/contracts/DepositReceipt_Base.sol#L164-L176 https://etherscan.io/address/0xc7de7f4d4C9c991fF62a07D18b3E31e349833A18#code https://etherscan.io/address/0x72002129A3834d63C57d157DDF069deE37b08F24#code

Tool used

Manual Review

Recommendation

Consider getting latest priceFeed.aggregator().minAnswer()/maxAnswer() in priceLiquidity()

Discussion

kree-dotcom

Sponsor confirmed, will fix.

Chainlink documents state: "you can call functions on the aggregator directly, but it is a best practice to use the AggregatorV3Interface to run functions on the proxy instead so that changes to the aggregator do not affect your application. Read the aggregator contract only if you need functions that are not available in the proxy."

So the auditor is right that we should not assume the AccessControlledOffchainAggregator is static. We will move these calls to occur on every call rather than in setup.

kree-dotcom

Fixed https://github.com/kree-dotcom/Velo-Deposit-Tokens/commit/58b8f3e14b416630971b7b17b500bbe22d2016aa

Note there are two fixes in this commit relating to the priceLiquidity function. the other issue is #72 , the code for these changes doesn't overlap so should be clear, please ask me if it is not.

Issue M-12: Vault_Synths.sol code does not consider protocol exchange fee when evaluating the Collateral worth

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/120

Found by

ctf_sec

Summary

Vault_Synths.sol code does not consider protocol fee.

Vulnerability Detail

If we look into the good-written documentation:

https://github.com/kree-dotcom/isomorph/blob/789338c8979ab75b8187781a2500908bb26dcdea/docs/Vault_Lyra.md#getwithdrawalfee

I want to quote:

> Because the withdrawalFee of a lyra LP pool can vary we must fetch it each time it is needed to ensure we use an accurate value. LP tokens are devalued by this as a safety measure as any liquidation would include selling the collateral and so should factor in that cost to ensure it is profitable.

In Vault_Lyra.sol, when calculating the collateral of the LP token, the fee is taken into consideration.

```
function priceCollateralToUSD(bytes32 _currencyKey, uint256 _amount) public view
↪   override returns(uint256){
     //The LiquidityPool associated with the LP Token is used for pricing
    ILiquidityPoolAvalon LiquidityPool =
↪   ILiquidityPoolAvalon(collateralBook.liquidityPoolOf(_currencyKey));
    //we have already checked for stale greeks so here we call the basic price
↪   function.
    uint256 tokenPrice = LiquidityPool.getTokenPrice();
    uint256 withdrawalFee = _getWithdrawalFee(LiquidityPool);
    uint256 USDValue  = (_amount * tokenPrice) / LOAN_SCALE;
    //we remove the Liquidity Pool withdrawalFee
    //as there's no way to remove the LP position without paying this.
    uint256 USDValueAfterFee = USDValue * (LOAN_SCALE- withdrawalFee)/LOAN_SCALE;
    return(USDValueAfterFee);
}
```

This is not the case for Vault_Synths.sol, the underlying token also charge exchange fee, but this fee is not reflected when evaluating the Collateral worth.

https://docs.synthetix.io/incentives/#exchange-fees

> Exchange fees are generated whenever a user exchanges one synthetic asset (Synth) for another through Synthetix.Exchange. Fees are typically between 10-100 bps (0.1%-1%), though usually 30 bps, and when generated are sent to the fee pool, where it is available to be claimed proportionally by SNX stakers each week.

If we go to https://synthetix.io/synths,

we can see that the sETH token charges 0.25%, the sBTC token charges 0.25%, the sUSD charges 0% fee, but this does not ensure this fee rate will not change in the future.

Impact

The collateral may be overvalued because the exchange does not count when evaluating the Collateral worth and result in bad debt which makes the project insolvent.

Code Snippet

Tool used

Manual Review

Recommendation

We recommend the project consider protocol exchange fee when evaluating the Collateral worth in Vault_Synths.sol

https://github.com/Synthetixio/synthetix/blob/develop/contracts/SystemSettings.sol#L362

Precisely when the exchange fee is updated, the fee is reflected in the collateral worth.

```
function setExchangeFeeRateForSynths(bytes32[] calldata synthKeys, uint256[]
↪   calldata exchangeFeeRates)
    external
    onlyOwner
{
    flexibleStorage().setExchangeFeeRateForSynths(SETTING_EXCHANGE_FEE_RATE,
↪   synthKeys, exchangeFeeRates);
    for (uint i = 0; i < synthKeys.length; i++) {
        emit ExchangeFeeUpdated(synthKeys[i], exchangeFeeRates[i]);
```

```
    }
}

/// @notice Set exchange dynamic fee threshold constant in decimal ratio
/// @param threshold The exchange dynamic fee threshold
/// @return uint threshold constant
function setExchangeDynamicFeeThreshold(uint threshold) external onlyOwner {
    require(threshold != 0, "Threshold cannot be 0");

    flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME,
↪   SETTING_EXCHANGE_DYNAMIC_FEE_THRESHOLD, threshold);

    emit ExchangeDynamicFeeThresholdUpdated(threshold);
}
```

Discussion

kree-dotcom

SPonsor confirmed, will fix

Issue M-13: All collateral in Velodrome vault will be permantly locked if either asset in liquidity pair stays outside of min/max price

Source: https://github.com/sherlock-audit/2022-11-isomorph-judging/issues/70

Found by

0x52, clems4ever

Summary

The oracles used have a built in safeguard to revert the transaction if the queried asset is outside of a defined price range. The issue with this is that every vault interaction requires the underlying collateral to be valued. If one of the assets in the pair goes outside it's immutable range then the entire vault will be frozen and all collateral will be permanently stuck.

Vulnerability Detail

```
function getOraclePrice(IAggregatorV3 _priceFeed, int192 _maxPrice, int192
 ↪  _minPrice) public view returns (uint256 ) {
    (
        /*uint80 roundID*/,
        int signedPrice,
        /*uint startedAt*/,
        uint timeStamp,
        /*uint80 answeredInRound*/
    ) = _priceFeed.latestRoundData();
    //check for Chainlink oracle deviancies, force a revert if any are present.
     ↪  Helps prevent a LUNA like issue
    require(signedPrice > 0, "Negative Oracle Price");
    require(timeStamp >= block.timestamp - HEARTBEAT_TIME , "Stale pricefeed");

    //@audit revert if price is outside of immutable bounds
    require(signedPrice < _maxPrice, "Upper price bound breached");
    require(signedPrice > _minPrice, "Lower price bound breached");
    uint256 price = uint256(signedPrice);
    return price;
}
```

The lines above are called each time and asset is priced. If the oracle returns outside of the predefined range then the transaction will revert.

```
uint256 outstandingisoUSD = isoUSDdebt - _USDToVault;
//@audit contract prices withdraw collateral
uint256 colInUSD = _calculateProposedReturnedCapital(_collateralAddress,
↪  _loanNFTs, _partialPercentage);
if(outstandingisoUSD >= TENTH_OF_CENT){ //ignore debts less than $0.001
    uint256 collateralLeft = totalCollateralValue(_collateralAddress,
    ↪  msg.sender) - colInUSD;
    uint256 borrowMargin = (outstandingisoUSD * minOpeningMargin) / LOAN_SCALE;
    require(collateralLeft > borrowMargin , "Remaining debt fails to meet
    ↪  minimum margin!");
}
```

When closing a loan the vault attempts to price the users collateral. Since this is the only way for a user to remove collateral is to call closeLoan, if the price of either asset in the LP goes outside of its bounds then all user deposits will be lost.

Impact

Entire vault will be frozen and all collateral will be permanently stuck

Code Snippet

https://github.com/sherlock-audit/2022-11-isomorph/blob/main/contracts/Isomorph/contracts/Vault_Velo.sol#L527-L587

Tool used

Manual Review

Recommendation

If a user is closing their entire loan then there is no need to check the value of the withdraw collateral because there is no longer any debt to collateralize. Move the check inside the inequality to allow the closeLoan to always function:

```
    uint256 outstandingisoUSD = isoUSDdebt - _USDToVault;
-   uint256 colInUSD = _calculateProposedReturnedCapital(_collateralAddress,
↪  _loanNFTs, _partialPercentage);
+   uint256 colInUSD;
    if(outstandingisoUSD >= TENTH_OF_CENT){ //ignore debts less than $0.001
+       uint256 colInUSD = _calculateProposedReturnedCapital(_collateralAddress,
↪  _loanNFTs, _partialPercentage);
        uint256 collateralLeft = totalCollateralValue(_collateralAddress,
        ↪  msg.sender) - colInUSD;
```

```
        uint256 borrowMargin = (outstandingisoUSD * minOpeningMargin) /
        ↪  LOAN_SCALE;
        require(collateralLeft > borrowMargin , "Remaining debt fails to meet
        ↪  minimum margin!");
    }
```

Discussion

kree-dotcom

Sponsor confirmed, will fix.

Also by fixing issue #145 and refreshing the minPrice and maxPrice on each call we will reduce the likelihood of a price getting stuck outside of the aggregator price range.

kree-dotcom

Proposed solution no longer works as the solution to issue #161 changed line 553 in `Vault_Velo.sol` to `if((outstandingisoUSD > 0) && (colInUSD > 0)){` meaning that the calculation of collnUSD must occur before entering the if clause.

However with issue #145 fixed this problem is less terminal as now we update the minPrice and maxPrice with each call. We then have two situations in which the minPrice or maxPrice protection is triggered:

1. Slow approach, i.e. like ETH reaching $1,000,000 over the span of a year. It is likely that Chainlink would update the aggregator maxPrice before it was reached and so this situation does not pose a problem.
2. Flash crash, a collateral is hacked or manipulated so that it drops below the minPrice faster than Chainlink can react and so the collateral is now stuck for our system. While this situation is not solved the assets backing the loan are now likely worthless anyway. This does not address the situation when the price rises suddenly. In that case the collateral will become inaccessible for a short while, it is assumed that if the price persists Chainlink would then alter the maxPrice and so collateral would become accessible again.

Therefore this has become a "won't fix" issue