SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	RealWagmi
Prepared by:	Sherlock
Lead Security Expert:	0x52
Dates Audited:	October 16 - October 23, 2023
Prepared on:	November 20, 2023

Introduction

Unlock the power of DeFi with Wagmi - an all-in-one platform for trading, liquidity provision, swapping, and yield strategy generation.

Scope

Repository: RealWagmi/wagmi-leverage

Branch: main

Commit: 21a66a01238aec1f1ecd14f4b3816c05ce37e1fe

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
8	4

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0x52
HHK
0xDetermination
handsomegiraffe
Bauer
ArmedGoose
detectiveking
0xJuda
talfao
mstpr-brainbot
shogoki
tsvetanovv

ali_shehab
seeques
Nyx
p-tsanev
0xMaroutis
0xpep7
0xMAKEOUTHILL
jah
kutugu
zraxx
AuditorPraise
lil.eth
lucifero
Japy69
0xReiAyanami
0xblackskull
Kral01
IceBear
peanuts
kaysoft
MohammedRizwan
pks_

SHERLOCK

Issue H-1: old borrowing key is used instead of `newBorrowingKey` when adding old loans to the newBorrowing in Liquidity-BorrowingManager.takeOverDebt()

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/53

Found by

0xMAKEOUTHILL, 0xMaroutis, 0xpep7, AuditorPraise, Nyx, ali_shehab, jah, kutugu, p-tsanev, seeques, zraxx when `_addKeysAndLoansInfo()` is called within LiquidityBorrowingManager.takeOverDebt(), old Borrowing Key is used and not `newBorrowingKey` see here

Vulnerability Detail

The old borrowing key credentials are deleted in `_removeKeysAndClearStorage(oldBorrowing.borrower, borrowingKey, oldLoans);` see here

And a new borrowing key is created with the holdToken, saleToken, and the address of the user who wants to take over the borrowing in the `_initOrUpdateBorrowing()`. see here

now the old borrowing key whose credentials are already deleted is used to update the old loans in `_addKeysAndLoansInfo()` instead of the `newBorrowingKey` generated in `_initOrUpdateBorrowing()` see here

Impact

wrong borrowing Key is used (i.e the old borrowing key) when adding old loans to `newBorrowing`

Therefore the wrong borrowing key (i.e the old borrowing key) will be added as borrowing key for tokenId of old Loans in `tokenIdToBorrowingKeys` in _addKeysAndLoansInfo()

(i.e when the bug of `update bool` being false, is corrected, devs should understand :))

Code Snippet

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L440-L441

Tool used

Manual Review

Recommendation

use newBorrowingKey when calling `_addKeysAndLoansInfo()` instead of old borrowing key.

Discussion

fann95

This is an inattention error related to the accelerated development of the project. I think we would have fixed this during testing, but thanks for the hint anyway. We will fix it.

fann95

Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/f5860c819eeb5146cc0c5c3b563ee83feb4ab33d

IAm0x52

Fix looks good. Correct variable is used now

Issue H-2: Adversary can reenter takeOverDebt() during liquidation to steal vault funds

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/76

Found by

0x52

Due to the lack of nonReentrant modifier on takeOverDebt() a liquidatable position can be both liquidated and transferred simultaneously. This results in LPs being repaid from the vault while the position and loans continue to be held open, effectively duplicating the liquidated position. LPs therefore get to 'double dip' from the vault, stealing funds and causing a deficit. This can be abused by an attacker who borrows against their own LP to exploit the 'double dip' for profit.

Vulnerability Detail

First we'll walk through a high level breakdown of the issue to have as context for the rest of the report:

1) Create a custom token that allows them to take control of the transaction and to prevent liquidation
2) Fund UniV3 LP with target token and custom token
3) Borrow against LP with target token as the hold token
4) After some time the position become liquidatable
5) Begin liquidating the position via repay()
6) Utilize the custom token during the swap in repay() to gain control of the transaction
7) Use control to reenter into takeOverDebt() since it lack nonReentrant modifier
8) Loan is now open on a secondary address and closed on the initial one
9) Transaction resumes (post swap) on repay()
10) Finish repayment and refund all initial LP
11) Position is still exists on new address
12) After some time the position become liquidatable
13) Loan is liquidated and attacker is paid more LP
14) Vault is at a deficit due to refunding LP twice
15) Repeat until the vault is drained of target token

LiquidityManager.sol#L279-L287

```
_v3SwapExactInput(
    v3SwapExactInputParams({
        fee: params.fee,
        tokenIn: cache.holdToken,
        tokenOut: cache.saleToken,
        amountIn: holdTokenAmountIn,
        amountOutMinimum: (saleTokenAmountOut * params.slippageBP1000) /
            Constants.BPS
    })
```

The control transfer happens during the swap to UniV3. Here when the custom token is transferred, it gives control back to the attacker which can be used to call takeOverDebt().

LiquidityBorrowingManager.sol#L667-L672

```
_removeKeysAndClearStorage(borrowing.borrower, params.borrowingKey, loans);
// Pay a profit to a msg.sender
_pay(borrowing.holdToken, address(this), msg.sender, holdTokenBalance);
_pay(borrowing.saleToken, address(this), msg.sender, saleTokenBalance);

emit Repay(borrowing.borrower, msg.sender, params.borrowingKey);
```

The reason the reentrancy works is because the actual borrowing storage state isn't modified until AFTER the control transfer. This means that the position state is fully intact for the takeOverDebt() call, allowing it to seamlessly transfer to another address behaving completely normally. After the repay() call resumes, _removeKeysAndClearStorage is called with the now deleted borrowKey.

Keys.sol#L31-L42

```
function removeKey(bytes32[] storage self, bytes32 key) internal {
    uint256 length = self.length;
    for (uint256 i; i < length; ) {
        if (self.unsafeAccess(i).value == key) {
            self.unsafeAccess(i).value = self.unsafeAccess(length - 1).value;
            self.pop();
            break;
        }
        unchecked {
            ++i;
        }
    }
```

The unique characteristic of deleteKey is that it doesn't revert if the key doesn't

exist. This allows "removing" keys from an empty array without reverting. This allows the repay call to finish successfully.

LiquidityBorrowingManager.sol#L450-L452

```
//newBorrowing.accLoanRatePerSeconds = oldBorrowing.accLoanRatePerSeconds;
_pay(oldBorrowing.holdToken, msg.sender, VAULT_ADDRESS, collateralAmt +
↪  feesDebt);
emit TakeOverDebt(oldBorrowing.borrower, msg.sender, borrowingKey,
↪  newBorrowingKey);
```

Now we can see how this creates a deficit in the vault. When taking over an existing debt, the user is only required to provide enough hold token to cover any fee debt and any additional collateral to pay fees for the newly transferred position. This means that the user isn't providing any hold token to back existing LP.

LiquidityBorrowingManager.sol#L632-L636

```
Vault(VAULT_ADDRESS).transferToken(
    borrowing.holdToken,
    address(this),
    borrowing.borrowedAmount + liquidationBonus
);
```

On the other hand repay transfers the LP backing funds from the vault. Since the same position is effectively liquidated twice, it will withdraw twice as much hold token as was originally deposited and no new LP funds are added when the position is taken over. This causes a deficit in the vault since other users funds are being withdrawn from the vault.

Impact

Vault can be drained

Code Snippet

LiquidityBorrowingManager.sol#L395-L453

Tool used

Manual Review

Recommendation

Add the `nonReentrant` modifier to `takeOverDebt()`

Discussion

fann95

Unfortunately, during development, we lost the nonReentrant-modifier just like checkDeadline. We will fix it.

fann95

Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/955d742c37736192f81a20c39a82324f3d711fb4

IAm0x52

Fix looks good. Nonreentrant has been added to takeOverDebt()

Issue H-3: Creditor can maliciously burn UniV3 position to permanently lock funds

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/78

Found by

0x52, 0xJuda, ArmedGoose, Bauer, HHK, handsomegiraffe, mstpr-brainbot, talfao

LP NFT's are always controlled by the lender. Since they maintain control, malicious lenders have the ability to burn their NFT. Once a specific tokenID is burned the ownerOf(tokenID) call will always revert. This is problematic as all methodologies to repay (even emergency) require querying the ownerOf() every single token. Since this call would revert for the burned token, the position would be permanently locked.

Vulnerability Detail

NonfungiblePositionManager

```
function ownerOf(uint256 tokenId) public view virtual override returns (address)
 ↪  {
    return _tokenOwners.get(tokenId, "ERC721: owner query for nonexistent
     ↪  token");
}
```

When querying a nonexistent token, ownerOf will revert. Now assuming the NFT is burnt we can see how every method for repayment is now lost.

LiquidityManager.sol#L306-L308

```
address creditor = underlyingPositionManager.ownerOf(loan.tokenId);
// Increase liquidity and transfer liquidity owner reward
_increaseLiquidity(cache.saleToken, cache.holdToken, loan, amount0, amount1);
```

If the user is being liquidated or repaying themselves the above lines are called for each loan. This causes all calls of this nature to revert.

LiquidityBorrowingManager.sol#L727-L732

```
for (uint256 i; i < loans.length; ) {
    LoanInfo memory loan = loans[i];
    // Get the owner address of the loan's token ID using the
     ↪  underlyingPositionManager contract.
    address creditor = underlyingPositionManager.ownerOf(loan.tokenId);
```

```
    // Check if the owner of the loan's token ID is equal to the `msg.sender`.
    if (creditor == msg.sender) {
```

The only other option to recover funds would be for each of the other lenders to call for an emergency withdrawal. The problem is that this pathway will also always revert. It cycles through each loan causing it to query ownerOf() for each token. As we know this reverts. The final result is that once this happens, there is no way possible to close the position.

Impact

Creditor can maliciously lock all funds

Code Snippet

LiquidityBorrowingManager.sol#L532-L674

Tool used

Manual Review

Recommendation

I would recommend storing each initial creditor when a loan is opened. Add try-catch blocks to each `ownerOf()` call. If the call reverts then use the initial creditor, otherwise use the current owner.

Discussion

fann95

Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/788c72b8242cc704e5db58cfdbfcdf8d4559d4b5

IAm0x52

Fix looks good. Burned tokens no longer cause a revert and tokens are sent to borrower as profit

Issue H-4: No slippage protection during repayment due to dynamic slippage params and easily influenced `slot0()`

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/109

Found by

0x52, 0xJuda, 0xMaroutis, 0xblackskull, HHK, IceBear, Kral01, MohammedRizwan, Nyx, kaysoft, lil.eth, lucifero, p-tsanev, peanuts, pks_, talfao, tsvetanovv The repayment function lacks slippage protection. It relies on slot0() to calculate sqrtLimitPrice, which in turn determines amounts for restoring liquidation. The dynamic calculation of slippage parameters based on these values leaves the function without adequate slippage protection, potentially reducing profit for the repayer.

Vulnerability Detail

The absence of slippage protection can be attributed to two key reasons. Firstly, the `sqrtPrice` is derived from `slot0()`, **which can be easily manipulated:**

```
 function _getCurrentSqrtPriceX96(
    bool zeroForA,
    address tokenA,
    address tokenB,
    uint24 fee
) private view returns (uint160 sqrtPriceX96) {
    if (!zeroForA) {
        (tokenA, tokenB) = (tokenB, tokenA);
    }
    address poolAddress = computePoolAddress(tokenA, tokenB, fee);
    (sqrtPriceX96, , , , , , ) = IUniswapV3Pool(poolAddress).slot0();
↪   //@audit-issue can be easily manipulated
}
```

The calculated `sqrtPriceX96` is used to determine the amounts for restoring liquidation and the number of holdTokens to be swapped for saleTokens:

```
(uint256 holdTokenAmountIn, uint256 amount0, uint256 amount1) =
↪   _getHoldTokenAmountIn(
                params.zeroForSaleToken,
                cache.tickLower,
                cache.tickUpper,
                cache.sqrtPriceX96,
                loan.liquidity,
                cache.holdTokenDebt
```

```
            );
```

After that, the number of `SaleTokemAmountOut` is gained based on the sqrtPrice via QuoterV2.

Then, the slippage params are calculated `amountOutMinimum: (saleTokenAmountOut * params.slippageBP1000) / Constants.BPS })` However, the `saleTokenAmountOut` is a dynamic number calculated on the current state of the blockchain, based on the calculations mentioned above. This will lead to the situation that the swap will always satisfy the `amountOutMinimum`.

As a result, if the repayment of the user is sandwiched (frontrunned), the profit of the repayer is decreased till the repayment satisfies the restored liquidity.

Proof of concept

A Proof of Concept (PoC) demonstrates the issue with comments. Although the swap does not significantly impact a strongly founded pool, it does result in a loss of a few dollars for the repayer.

```
        let amountWBTC = ethers.utils.parseUnits("0.05", 8); //token0
         const deadline = (await time.latest()) + 60;
         const minLeverageDesired = 50;
         const maxCollateralWBTC = amountWBTC.div(minLeverageDesired);

         const loans = [
             {
                 liquidity: nftpos[3].liquidity,
                 tokenId: nftpos[3].tokenId,
             },
         ];

         const swapParams: ApproveSwapAndPay.SwapParamsStruct = {
             swapTarget: constants.AddressZero,
             swapAmountInDataIndex: 0,
             maxGasForCall: 0,
             swapData: swapData,
         };

let  params = {
             internalSwapPoolfee: 500,
             saleToken: WETH_ADDRESS,
             holdToken: WBTC_ADDRESS,
             minHoldTokenOut: amountWBTC,
             maxCollateral: maxCollateralWBTC,
             externalSwap: swapParams,
             loans: loans,
```

```
        };

await borrowingManager.connect(bob).borrow(params, deadline);

const borrowingKey = await borrowingManager.userBorrowingKeys(bob.address, 0);
        const swapParamsRep: ApproveSwapAndPay.SwapParamsStruct = {
            swapTarget: constants.AddressZero,
            swapAmountInDataIndex: 0,
            maxGasForCall: 0,
            swapData: swapData,
        };


        amountWBTC = ethers.utils.parseUnits("0.06", 8); //token0

let swapping: ISwapRouter.ExactInputSingleParamsStruct = {
            tokenIn: WBTC_ADDRESS,
            tokenOut: WETH_ADDRESS,
            fee: 500,
            recipient: alice.address,
            deadline: deadline,
            amountIn: ethers.utils.parseUnits("100", 8),
            amountOutMinimum: 0,
            sqrtPriceLimitX96: 0
        };
        await router.connect(alice).exactInputSingle(swapping);
        console.log("Swap success");

 let paramsRep: LiquidityBorrowingManager.RepayParamsStruct = {
            isEmergency: false,
            internalSwapPoolfee: 500,
            externalSwap: swapParamsRep,
            borrowingKey: borrowingKey,
            swapSlippageBP1000: 990, //<=slippage simulated
        };
 await borrowingManager.connect(bob).repay(paramsRep, deadline);
        // Without swap
// Balance of hold token after repay:  BigNumber { value: "993951415" }
// Balance of sale token after repay:  BigNumber { value: "99005137946252426108"
↪  }
// When swap
// Balance of hold token after repay:  BigNumber { value: "993951415" }
// Balance of sale token after repay:  BigNumber { value: "99000233164653177505"
↪  }
```

The following table shows difference of recieved sale token: | Swap before repay transaction | Token | Balance of user after Repay |

|-------------------------|--------------------|----------------------| | No | WETH | 99005137946252426108 | | Yes | WETH |99000233164653177505 |

The difference in the profit after repayment is 4904781599248603 weis, which is at the current market price of around 8 USD. The profit loss will depend on the liquidity in the pool, which depends on the type of pool and related tokens.

Impact

The absence of slippage protection results in potential profit loss for the repayer.

Code Snippet

Slot0 is used here Dynamic slippage params are created here - saleTokenAmount is dynamic variable calculated on the state of blockchain.

Tool used

Manual Review

Recommendation

To address this issue, avoid relying on slot0 and instead utilize Uniswap TWAP. Additionally, consider manually setting values for amountOutMin for swaps based on data acquired before repayment.

Discussion

fann95

I think the severity level is medium since exchange within the pool is not the only way, there is also an external swap.

fann95

Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/bf84623009654acee8ee26ba2805068d2806e745

Czar102

> I think the severity level is medium since exchange within the pool is not the only way, there is also an external swap.

Even though it concerns only a subset of the functionalities of the protocol, the impact seems to be of high severity. Please let me know if that's not the case.

fann95

> I think the severity level is medium since exchange within the pool is not the only way, there is also an external swap.

> Even though it concerns only a subset of the functionalities of the protocol, the impact seems to be of high severity. Please let me know if that's not the case.

I leave the decision up to you.

Czar102

I'll leave it high severity because of the reason mentioned above.

IAm0x52

Fix looks good. sqrtPriceLimitX96 is now used to determine if a swap has been frontrun. If the simulated swap results in a higher (or lower for sells) price limit than expected the transaction will revert.

Issue M-1: DoS of lenders and gas griefing by packing tokenIdToBorrowingKeys arrays

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/15

Found by

0xDetermination In `LiquidityBorrowingManager`, `tokenIdToBorrowingKeys` arrays can be packed to gas grief and cause DoS of specific loans for an arbitrary period of time.

Vulnerability Detail

`LiquidityBorrowingManager.borrow()` calls the function `_addKeysAndLoansInfo()`, which adds user keys to the `tokenIdToBorrowingKeys` array of the borrowed-from LP position:

```
function _addKeysAndLoansInfo(
    bool update,
    bytes32 borrowingKey,
    LoanInfo[] memory sourceLoans
) private {
    // Get the storage reference to the loans array for the borrowing key
    LoanInfo[] storage loans = loansInfo[borrowingKey];
    // Iterate through the sourceLoans array
    for (uint256 i; i < sourceLoans.length; ) {
        // Get the current loan from the sourceLoans array
        LoanInfo memory loan = sourceLoans[i];
        // Get the storage reference to the tokenIdLoansKeys array for the
↪   loan's token ID
        bytes32[] storage tokenIdLoansKeys =
↪   tokenIdToBorrowingKeys[loan.tokenId];
        // Conditionally add or push the borrowing key to the tokenIdLoansKeys
↪   array based on the 'update' flag
        update
            ? tokenIdLoansKeys.addKeyIfNotExists(borrowingKey)
            : tokenIdLoansKeys.push(borrowingKey);
...
```

A user key is calculated in the `Keys` library like so:

```
function computeBorrowingKey(
    address borrower,
    address saleToken,
    address holdToken
) internal pure returns (bytes32) {
```

```
    return keccak256(abi.encodePacked(borrower, saleToken, holdToken));
}
```

So every time a new user borrows some amount from a LP token, a new `borrowKey` is added to the `tokenIdToBorrowingKeys[LP_Token_ID]` array. The problem is that this array is iterated through by calling iterating methods (`addKeyIfNotExists()` or `removeKey()`) in the `Keys` library when updating a borrow (as seen in the first code block). Furthermore, emergency repays call `removeKey()` in `_calculateEmergencyLoanClosure()`, non-emergency repays call `removeKey()` in `_removeKeysAndClearStorage()`, and `takeOverDebt()` calls `removeKey()` in `_removeKeysAndClearStorage()`. The result is that all exit/repay/liquidation methods must iterate through the array. Both of the iterating methods in the `Keys` library access storage to compare array values to the key passed as argument, so every key in the array before the argument key will increase the gas cost of the transaction by (more than) a cold `SLOAD`, which costs 2100 gas (https://eips.ethereum.org/EIPS/eip-2929). Library methods below:

```
function addKeyIfNotExists(bytes32[] storage self, bytes32 key) internal {
    uint256 length = self.length;
    for (uint256 i; i < length; ) {
        if (self.unsafeAccess(i).value == key) {
            return;
        }
        unchecked {
            ++i;
        }
    }
    self.push(key);
}

function removeKey(bytes32[] storage self, bytes32 key) internal {
    uint256 length = self.length;
    for (uint256 i; i < length; ) {
        if (self.unsafeAccess(i).value == key) {
            self.unsafeAccess(i).value = self.unsafeAccess(length - 1).value;
            self.pop();
            break;
        }
        unchecked {
            ++i;
        }
    }
}
```

Let's give an example to see the potential impact and cost of the attack:

1. An LP provider authorizes the contract to give loans from their large position.

Let's say USDC/WETH pool.

2. The attacker sees this and takes out minimum borrows of USDC using different addresses to pack the position's `tokenIdToBorrowingKeys` array. In `Constants.sol`, `MINIMUM_BORROWED_AMOUNT = 100000` so the minimum borrow is $0.1 dollars since USDC has 6 decimal places. Add this to the estimated gas cost of the borrow transaction, let's say $3.9 dollars. The cost to add one key to the array is approx. $4. The max block gas limit on ethereum mainnet is `30,000,000`, so divide that by 2000 gas, the approximate gas increase for one key added to the array. The result is 15,000, therefore the attacker can spend 60000 dollars to make any new borrows from the LP position unable to be repaid, transferred, or liquidated. Any new borrow will be stuck in the contract.
3. The attacker now takes out a high leverage borrow on the LP position, for example $20,000 in collateral for a $1,000,000 borrow. The attacker's total expenditure is now $80,000, and the $1,000,000 from the LP is now locked in the contract for an arbitrary period of time.
4. The attacker calls `increaseCollateralBalance()` on all of the spam positions. Default daily rate is .1% (max 1%), so over a year the attacker must pay 36.5% of each spam borrow amount to avoid liquidation and shortening of the array. If the gas cost of increasing collateral is $0.5 dollars, and the attacker spends another $0.5 dollars to increase collateral for each spam borrow, then the attacker can spend $1 on each spam borrow and keep them safe from liquidation for over 10 years for a cost of $15,000 dollars. The total attack expenditure is now $95,000. The protocol cannot easily increase the rate to hurt the attacker, because that would increase the rate for all users in the USDC/WETH market. Furthermore, the cost of the attack will not increase that much even if the daily rate is increased to the max of 1%. The attacker does not need to increase the collateral balance of the $1,000,000 borrow since repaying that borrow is DoSed.
5. The result is that $1,000,000 of the loaner's liquidity is locked in the contract for over 10 years for an attack cost of $95,000.

Impact

Array packing causes users to spend more gas on loans of the affected LP token. User transactions may out-of-gas revert due to increased gas costs. An attacker can lock liquidity from LPs in the contract for arbitrary periods of time for asymmetric cost favoring the attacker. The LP will earn very little fees over the period of the DoS.

Code Snippet

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L100-L101

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L790-L826 https://github.com/sherlock-audit/2023-10-real-wagmi/blob/main/wagmi-leverage/contracts/libraries/Keys.sol

Tool used

Manual Review

Recommendation

`tokenIdToBorrowingKeys` tracks borrowing keys and is used in view functions to return info (getLenderCreditsCount() and getLenderCreditsInfo()). This functionality is easier to implement with arrays, but it can be done with mappings to reduce gas costs and prevent gas griefing and DoS attacks. For example the protocol can emit the borrows for all LP tokens and keep track of them offchain, and pass borrow IDs in an array to a view function to look them up in the mapping. Alternatively, OpenZeppelin's EnumerableSet library could be used to replace the array and keep track of all the borrows on-chain.

Discussion

fann95

Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/cb4d91fdc632b1a4496932ec4546c7f4fa78e842

IAm0x52

Fix looks good. `tokenIdToBorrowingKeys` has been changed to an enumerable set removing the gas DOS

Issue M-2: No deadline and slippage check on `takeOverDebt()` can lead to unexpected results

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/51

Found by

HHK

The function takeOverDebt() in the LiquidityBorrowingManager contract doesn't have any deadline check like borrow or repay.

Additionally it also doesn't have any "slippage" check that would make sure the position hasn't changed between the moment a user calls takeOverDebt() and the transaction is confirmed.

Vulnerability Detail

Blockchains are asynchronous by nature, when sending a transaction, the contract targeted can see its state changed affecting the result of our transaction.

When a user wants to call takeOverDebt() he expects that the debt he is gonna take over will be the same (or very close) as when he signed the transaction.

By not providing any deadline check, if the transaction is not confirmed for a long time then the user might end up with a position that is not as interesting as it was.

Take this example:

- A position on Ethereum is in debt of collateral but the borrowed tokens are at profit and the user think the token's price is going to keep increasing, it makes sense to take over.
- The user makes a transaction to take over.
- The gas price rise and the transaction takes longer than he thoughts to be confirmed.
- Eventually the transaction is confirmed but the position is not in profit anymore because price changed during that time.
- User paid the debt of the position for nothing as he won't be making profits.

Additionally when the collateral of a position is negative, lenders can call repay() as part of the "emergency liquidity restoration mode" which will reduce the size of the position. If this happens while another user is taking over the debt then he might end up with a position that is not as interesting as he thoughts.

Take this second example:

- A position on Ethereum with 2 loans is in debt of collateral but the borrowed tokens are at profit and the user think the token's price is going to keep increasing, it makes sense to take over.
- The user makes a transaction to take over.
- While the user's transaction is in the MEMPOOL a lender call 'repay()' and get back his tokens.
- The user's transaction is confirmed and he take over the position but it only has 1 loan now as one of the 2 loans was sent back to the lender. the position might not be at profit anymore or less than it was supposed to be.
- User paid the debt of the position for nothing as he won't be making profits.

Impact

Medium. User might pay collateral debt of a position for nothing.

Code Snippet

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/b33752757fd6a9f404b8577c1eae6c5774b3a0db/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L395

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/b33752757fd6a9f404b8577c1eae6c5774b3a0db/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L581

Tool used

Manual Review

Recommendation

Consider adding the modifier `checkDeadline()` as well as a parameter `minBorrowedAmount` and compare it to the current `borrowedAmount` to make sure no lender repaid their position during the take over.

Discussion

fann95

we will add a deadline check.

fann95

Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/97f30b9cf2ef1715f6050eb6b2d9d044235af86f

IAm0x52

Escalate

While I agree that having the deadline enforcement is ideal, I don't see how this would cause any negative impact to the user taking over the loan besides wasted gas. Taking over a loan can only be done for a position that is eligible for liquidation. Liquidation is highly incentivized which means that a delayed transaction would already be very unlikely to succeed.

In the event that the position is not changed at all, the new borrower has collateralAmt to protect themselves from excess fees. If the owner of the position repays or if the loan is liquidated by another user ahead of this transaction, the takeOverDebt call will revert. If a lender requests emergency closure ahead of time then the fees owed will be reduced proportionally and any excess underwater fees would be written off here during the emergency repayment.

Due to this I don't see any negative financial impact to the new borrower besides wasted gas. I believe this should be a valid low.

sherlock-admin2

> Escalate
>
> While I agree that having the deadline enforcement is ideal, I don't see how this would cause any negative impact to the user taking over the loan besides wasted gas. Taking over a loan can only be done for a position that is eligible for liquidation. Liquidation is highly incentivized which means that a delayed transaction would already be very unlikely to succeed.
>
> In the event that the position is not changed at all, the new borrower has collateralAmt to protect themselves from excess fees. If the owner of the position repays or if the loan is liquidated by another user ahead of this transaction, the takeOverDebt call will revert. If a lender requests emergency closure ahead of time then the fees owed will be reduced proportionally and any excess underwater fees would be written off here during the emergency repayment.
>
> Due to this I don't see any negative financial impact to the new borrower besides wasted gas. I believe this should be a valid low.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

Czar102

A user calling `takeOverDebt()` proposes to liquidate a position. Enforcing timely execution is callers' responsibility. Adding a deadline argument seems like a good add-on, but its lack does not seem to cause any loss of funds. As far as I understand, this function will be called by MEV extractors, so revert after repaying is not an issue. It's the risk they are taking by design. Realizing losses by liquidators because of price movement is out of scope. If there are other possible kinds of losses, please let me know. Otherwise will consider the issue a low and accept the escalation.

HHK-ETH

> If a lender requests emergency closure ahead of time then the fees owed will be reduced proportionally and any excess underwater fees would be written off here during the emergency repayment.

I believe this is wrong, the line tagged add the collateral that was available to use to the fees owned to the lenders but it doesn't write off the underwater fees. These underwater fees are saved here. By updating the `accLoanRatePerSeconds` the contract will be able to recalculate the missing collateral on future calls.

Thus if a lender use emergency repay the user taking over the debt will be repaying the full missing collateral for a position that might not be worth it anymore.

HHK-ETH

You can check this comment from another issue to illustrate my point: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/119#issuecomment-1784834772

As confirmed by the sponsor, the debt is maintained as no one is paying it off and to do that we play with the `accLoanRatePerSeconds`.

This means the user taking over pays the debt the position owned to all lenders including the ones that emergency withdrew. So we need to make sure no loan was emergency repaid during that time otherwise the user may end up at a loss by repaying the whole collateral missing for a position that is not worth it anymore.

My proposition was first to add a deadline but this is only half a fix, thus I proposed to also introduce a `minBorrowedAmount` parameter.

Czar102

@IAm0x52 could you verify the above comment?

> Thus if a lender use emergency repay the user taking over the debt will be repaying collateral for a position that might not be worth it.

@HHK-ETH that would mean that the emergency withdraw could make underwater positions more underwater? Could you define "might not be worth it"?

HHK-ETH

> @HHK-ETH that would mean that the emergency withdraw could make underwater positions more underwater? Could you define "might not be worth it"?

What I mean is that if you want to take over then you estimate that keeping the current position can make you money and thus you're okay to pay back the missing collateral. Let's take an example:

- A position has 5 ETH long over 2 different loans and 0.2 ETH of missing collateral
- User is bullish on ETH and wants to take over instead of liquidate the position. Take note that when liquidating you don't have to pay the missing collateral.
- During the take over, one of the loans is reimbursed taking away 3 ETH from the position.
- Once take over is confirmed, the position now holds only 2 ETH and not the initial 5 ETH. the user repaid 0.2 ETH but has a smaller position than expected and the long might not generate enough profits to reimburse the extra collateral paid (0.2 ETH).

If he knew it would have happened he maybe would have called `liquidate()` instead or `borrow()` to borrow from different positions that might not generate as much upside but at least wouldn't require to repay missing collateral.

HHK-ETH

> Take this second example:

This example used in the initial finding can be a little confusing as I talk about positions in profits. But after going through the contracts again I think the profits are not necessarily taken away when a lender emergency repays and so would stay on the position.

Still the example I gave above still stands, even in profits, if they're not superior than the underwater collateral paid (minus the `liquidationBonus` you got from the old position) you end up at a loss that you might not be able to reimburse longing a smaller position than expected.

IAm0x52

> > @HHK-ETH that would mean that the emergency withdraw could make underwater positions more underwater? Could you define "might not be worth it"?
>
> What I mean is that if you want to take over then you estimate that keeping the current position can make you money and thus you're okay to pay back the missing collateral. Let's take an example:
>
> - A position has 5 ETH long over 2 different loans and 0.2 ETH of missing collateral

- User is bullish on ETH and wants to take over instead of liquidate the position. Take note that when liquidating you don't have to pay the missing collateral.
- During the take over, one of the loans is reimbursed taking away 3 ETH from the position.
- Once take over is confirmed, the position now holds only 2 ETH and not the initial 5 ETH. the user repaid 0.2 ETH but has a smaller position than expected and the long might not generate enough profits to reimburse the extra collateral paid (0.2 ETH).

If he knew it would have happened he maybe would have called `liquidate()` instead or `borrow()` to borrow from different positions that might not generate as much upside but at least wouldn't require to repay missing collateral.

The problem with this is that the user taking over would not owe 0.2 ETH still because here fees owed by the position is reduced. So if the position owed 0.2 ETH and closed 3/5 (60%) then it would also remove 3/5 of the fees as we can see here since it uses a proportional calculation. This would leave the position with 0.08 ETH to repay when taken over.

Czar102

@HHK-ETH can you confirm?

HHK-ETH

> The problem with this is that the user taking over would not owe 0.2 ETH still because here fees owed by the position is reduced. So if the position owed 0.2 ETH and closed 3/5 (60%) then it would also remove 3/5 of the fees as we can see here since it uses a proportional calculation. This would leave the position with 0.08 ETH to repay when taken over.

But if the total collateral to be paid is below 0 the fees accounted here are only the dailyCollateral available not the missing collateral. I believe `feesOwed` is accounting only fees that have been paid and not fees that couldn't be paid because of missing collateral.

HHK-ETH

Take this poc you can copy paste it in the main test file just keep the setup and nft creation test and delete the rest:

```
it("Updated accRate", async () => {
        const amountWBTC = ethers.utils.parseUnits("0.05", 8); //token0
        let deadline = (await time.latest()) + 60;
        const minLeverageDesired = 50;
        const maxCollateralWBTC = amountWBTC.div(minLeverageDesired);
```

```
        const loans = [
            {
                liquidity: nftpos[3].liquidity,
                tokenId: nftpos[3].tokenId,
            },
            {
                liquidity: nftpos[5].liquidity,
                tokenId: nftpos[5].tokenId,
            },
        ];

        const swapParams: ApproveSwapAndPay.SwapParamsStruct = {
            swapTarget: constants.AddressZero,
            swapAmountInDataIndex: 0,
            maxGasForCall: 0,
            swapData: swapData,
        };

        const borrowParams = {
            internalSwapPoolfee: 500,
            saleToken: WETH_ADDRESS,
            holdToken: WBTC_ADDRESS,
            minHoldTokenOut: amountWBTC,
            maxCollateral: maxCollateralWBTC,
            externalSwap: swapParams,
            loans: loans,
        };

        //borrow tokens
        await borrowingManager.connect(bob).borrow(borrowParams, deadline);

        await time.increase(3600 * 25); //1h of missing collateral since when we
↪   borrow we add collateral for a day
        deadline = (await time.latest()) + 60;

        const borrowingKey = await
↪   borrowingManager.userBorrowingKeys(bob.address, 0);

        let repayParams = {
            isEmergency: true,
            internalSwapPoolfee: 0,
            externalSwap: swapParams,
            borrowingKey: borrowingKey,
            swapSlippageBP1000: 0,
        };

        const oldCollateralMissing = await
↪   borrowingManager.checkDailyRateCollateral(borrowingKey);
```

```
        console.log(oldCollateralMissing.balance);

        //Alice emergency repay her loan
        await borrowingManager.connect(alice).repay(repayParams, deadline);

        const newCollateralMissing = await
↪   borrowingManager.checkDailyRateCollateral(borrowingKey);
        console.log(newCollateralMissing.balance);

        //missing collateral is still the same
        expect(oldCollateralMissing.balance).to.eq(newCollateralMissing.balance);
    });
```

It shows that the missing collateral is still the same after emergency repay.

IAm0x52

> I believe this is wrong, the line tagged add the collateral that was available to use to the fees owned to the lenders but it doesn't write off the underwater fees. These underwater fees are saved here. By updating the `accLoanRatePerSeconds` the contract will be able to recalculate the missing collateral on future calls.

Yes this comment is incorrect. Underwater fees are saved. My initial comment regarding emergency closure was made based on the misconception that underwater fees are written off. Since they are not written off and fees continue to accumulate as normal minBorrowedAmount would still effectively function as a deadline. If the position is currently accumulating 1e16 interest per second then using a minBorrowedAmount that is 1e17 above the current amount owed, then if the takeOverDebt is delayed by more than 10 seconds then the call will fail.

HHK-ETH

See the finding recommendation:

> `minBorrowedAmount` and compare it to the current `borrowedAmount` to make sure no lender repaid their position during the take over

My proposition doesn't act as a deadline but as a check on the position's size, by checking `borrowingsInfo[key].borrowedAmount`. If it's below the passed parameter `minBorrowedAmount` then revert.

This effectively protects you from having lender repaying their position while your transaction is being confirmed.

IAm0x52

> See the finding recommendation:
>
> > `minBorrowedAmount` and compare it to the current `borrowedAmount` to make sure no lender repaid their position during the take over

> My proposition doesn't act as a deadline but as a check on the position's size, by checking `borrowingsInfo[key].borrowedAmount`. If it's below the passed parameter `minBorrowedAmount` then revert.

So then what exact loss scenario are you preventing? As shown above, there is already effectively a deadline. The only one I see would be for a lender to frontrun their call with an emergency closure. But what would be the benefit of that? The lender calling it would end up losing fees that it would have otherwise received. They could **potentially** cause a loss to the user taking over the loan but would guarantee a loss for themselves.

HHK-ETH

> So then what exact loss scenario are you preventing? As shown above, there is already effectively a deadline. The only one I see would be for a lender to frontrun their call with an emergency closure. But what would be the benefit of that? The lender calling it would end up losing fees that it would have otherwise received. They could potentially cause a loss to the user taking over the loan but would guarantee a loss for themselves.

The lender doesn't have to be malicious, he just need to be submitting a repay around the same time that the user is taking over. He doesn't know that someone is planning on taking over and might want his tokens back. Yes this is not likely to happen but it can happen if both are unlucky and will result in loss for the user and less fees for the lender.

I submitted this finding with slippage in the title because you could see it as a swap, let's imagine a nice world with no MEV. If you submit a swap on a pool but someone else swap on it before you (this user is not malicious but just happened to have called swap around the same time as you), you end up with a different amount than expected, Thus `minOut` act as a slippage protection. Here it's kind of the same but way less likely because a position can have only 7 loans maximum so there is only 7 other actors, **still the chance is not 0**.

Additionally like you said lender can go rogue and loose his fees to make user taking over pay collateral for a smaller position.

Since there is money at stake I submitted this as a medium (and it seemed to be fitting criteria 1 or 3 https://docs.sherlock.xyz/audits/judging/judging#v.-how-to-identify-a-medium-issue) but ultimately if you think it's not likely enough to happen and cost is too high for a malicious lender then I could understand that severity is lowered.

fann95

https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/119 I have made changes to the emergency repayment. Now, in case of an emergency repayment, the debts for this liquidity will also be written off. Therefore, if someone takes over a debt, he will not pay in the event of a frontrunning emergency

repayment.

Czar102

I think this issue is correctly classified as a medium. Planning to reject the escalation.

HHK-ETH

If only the slippage part of this finding is accepted as a medium and the deadline is considered low I would suggest reviewing the duplicate https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/134 as it only talks about deadline check.

Czar102

Planning to invalidate #134, too.

Evert0x

Based on the conversation this issue should be Medium without #134 as a duplicate.

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- IAm0x52: rejected

IAm0x52

Fix looks good. The `checkDeadline` modifier has been added. In conjuction with changes made for #119 this is no longer an issue.

Issue M-3: Adversary can overwrite function selector in _patchAmountAndCall due to inline assembly lack of over-flow protection

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/82

Found by

0x52

When using inline assembly, the standard overflow/underflow protections do not apply. This allows an adversary to specify a swapAmountInDataIndex which after multiplication and addition allows them to overwrite the function selector. Using a created token in a UniV3 LP pair they can manufacture any value for swapAmountInDataValue.

Vulnerability Detail

The use of YUL or inline assembly in a solidity smart contract also makes integer overflow/ underflow possible even if the compiler version of solidity is 0.8. In YUL programming language, integer underflow & overflow is possible in the same way as Solidity and it does not check automatically for it as YUL is a low-level language that is mostly used for making the code more optimized, which does this by omitting many opcodes. Because of its low-level nature, YUL does not perform many security checks therefore it is recommended to use as little of it as possible in your smart contracts.

Source

Inline assembly lacks overflow/underflow protections, which opens the possibility of this exploit.

ExternalCall.sol#L27-L38

```
if gt(swapAmountInDataValue, 0) {
    mstore(add(add(ptr, 0x24), mul(swapAmountInDataIndex, 0x20)),
    ↪  swapAmountInDataValue)
}
success := call(
    maxGas,
    target,
    0, //value
    ptr, //Inputs are stored at location ptr
    data.length,
    0,
```

```
    0
)
```

In the code above we see that `swapAmountInDataValue` is stored at `ptr + 36 (0x24) + swapAmountInDataIndex * 32 (0x20)`. The addition of 36 (0x24) in this scenario should prevent the function selector from being overwritten because of the extra 4 bytes (using 36 instead of 32). This is not the case though because `mul(swapAmountInDataIndex, 0x20)` can overflow since it is a uint256. This allows the attacker to target any part of the memory they choose by selectively overflowing to make it write to the desired position.

As shown above, overwriting the function selector is possible although most of the time this value would be a complete nonsense since swapAmountInDataValue is calculated elsewhere and isn't user supplied. This also has a work around. By creating their own token and adding it as LP to a UniV3 pool, swapAmountInDataValue can be carefully manipulated to any value. This allows the attacker to selectively overwrite the function selector with any value they chose. This bypasses function selectors restrictions and opens calls to dangerous functions.

Impact

Attacker can bypass function restrictions and call dangerous/unintended functions

Code Snippet

ExternalCall.sol#L14-L47

Tool used

Manual Review

Recommendation

Limit `swapAmountInDataIndex` to a reasonable value such as uint128.max, preventing any overflow.

Discussion

fann95

I don't think this is a realistic scenario. You won't be able to get the offset to the function selector by overflowing the results of the multiplication or addition here. Using which index can you get the mload(0x40) ?

Czar102

I think overwriting the selector could be done if the `swapAmountInDataIndex` was `k * 2 ** 251 - 2` for any integer `k`. This is because `swapAmountInDataIndex * 0x20 = k * 2 ** 251 * 2 ** 5 - 2 * 2 ** 5 = k * 2 ** 256 - 64 = - 64` mod `2 ** 256`. Changing memory at a pointer with such modification would collide with the selector at 4 least significant bytes and would write 28 bytes to previously allocated memory (memory corruption). My recommendation would be to limit `swapAmountInDataIndex` to `div(data.length, 0x20)`. Then, one would be able to write `swapAmountInDataValue` at any correct index in the calldata or right behind the calldata, if that amount is not to show up in the function call.

fann95

Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/b568b21ae07587ee784bba2c25c7562207a4b027

IAm0x52

Fix looks good. Indexes that indicate a position larger then the calldata size cause the function to revert.

Issue M-4: Blacklisted creditor can block all repayment besides emergency closure

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/83

Found by

0x52, ArmedGoose, Bauer, tsvetanovv

After liquidity is restored to the LP, accumulated fees are sent directly from the vault to the creditor. Some tokens, such as USDC and USDT, have blacklists the prevent users from sending or receiving tokens. If the creditor is blacklisted for the hold token then the fee transfer will always revert. This forces the borrower to defualt. LPs can recover their funds but only after the user has defaulted and they request emergency closure.

Vulnerability Detail

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/main/wagmi-leverage/contracts/abstract/LiquidityManager.sol#L306-L315

```
address creditor = underlyingPositionManager.ownerOf(loan.tokenId);
// Increase liquidity and transfer liquidity owner reward
_increaseLiquidity(cache.saleToken, cache.holdToken, loan, amount0, amount1);
uint256 liquidityOwnerReward = FullMath.mulDiv(
    params.totalfeesOwed,
    cache.holdTokenDebt,
    params.totalBorrowedAmount
) / Constants.COLLATERAL_BALANCE_PRECISION;

Vault(VAULT_ADDRESS).transferToken(cache.holdToken, creditor,
↪   liquidityOwnerReward);
```

The following code is executed for each loan when attempting to repay. Here we see that each creditor is directly transferred their tokens from the vault. If the creditor is blacklisted for holdToken, then the transfer will revert. This will cause all repayments to revert, preventing the user from ever repaying their loan and forcing them to default.

Impact

Borrowers with blacklisted creditors are forced to default

Code Snippet

LiquidityManager.sol#L223-L321

Tool used

Manual Review

Recommendation

Create an escrow to hold funds in the event that the creditor cannot receive their funds. Implement a try-catch block around the transfer to the creditor. If it fails then send the funds instead to an escrow account, allowing the creditor to claim their tokens later and for the transaction to complete.

Discussion

fann95

Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/3c17a39e8a69a8912e6f87e84a19f55889353328

IAm0x52

Fix looks good. Fee collection has been made generic instead of specific to protocol fees. Creditor fees are now cached and collected.

Issue M-5: Incorrect calculations of borrowingCollateral leads to DoS for positions in the current tick range due to underflow

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/86

Found by

0xReiAyanami, Bauer, Japy69, ali_shehab, lil.eth, lucifero, seeques The `borrowingCollateral` is the amount of collateral a borrower needs to pay for his leverage. It should be calculated as the difference of holdTokenBalance (the amount borrowed + holdTokens received after saleTokens are swapped) and the amount borrowed and checked against user-specified maxCollateral amount which is the maximum the borrower wishes to pay. However, in the current implementation the `borrowingCollateral` calculation is most likely to underflow.

Vulnerability Detail

This calculation is most likely to underflow

```
uint256 borrowingCollateral = cache.borrowedAmount - cache.holdTokenBalance;
```

The `cache.borrowedAmount` is the calculated amount of holdTokens based on the liquidity of a position. `cache.holdTokenBalance` is the balance of holdTokens queried after liquidity extraction and tokens transferred to the `LiquidityBorrowingManager`. If any amounts of the saleToken are transferred as well, these are swapped to holdTokens and added to `cache.holdTokenBalance`.

So in case when liquidity of a position is in the current tick range, both tokens would be transferred to the contract and saleToken would be swapped for holdToken and then added to `cache.holdTokenBalance`. This would make `cache.holdTokenBalance > cache.borrowedAmount` since `cache.holdTokenBalance == cache.borrowedAmount + amount of sale token swapped` and would make the tx revert due to underflow.

Impact

Many positions would be unavailable to borrowers. For non-volatile positions like that which provide liquidity to stablecoin pools the DoS could last for very long period. For volatile positions that provide liquidity in a wide range this could also be for more than 1 year.

Code Snippet

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L492-L503 https://github.com/sherlock-audit/2023-10-real-wagmi/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L470 https://github.com/sherlock-audit/2023-10-real-wagmi/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L848-L896

Tool used

Manual Review

Recommendation

The borrowedAmount should be subtracted from holdTokenBalance

```
uint256 borrowingCollateral = cache.holdTokenBalance - cache.borrowedAmount;
```

Discussion

Ali-Shehab

Escalate

First issue has nothing to do with the other ones. It must not be duplicate

[peanuts - Max collateral check is not done when increasing collateral balance] https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/37

sherlock-admin2

> Escalate
>
> First issue has nothing to do with the other ones. It must not be duplicate
>
> [peanuts - Max collateral check is not done when increasing collateral balance] https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/37

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

fann95

Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/7937e25a2d344881c7c4fb202ed89965b0fed229

IAm0x52

Escalate

This is a valid issue but it should be medium rather than high. Impact is broken functionality and DOS which doesn't qualify as high.

sherlock-admin2

> Escalate
>
> This is a valid issue but it should be medium rather than high. Impact is broken functionality and DOS which doesn't qualify as high.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

Ali-Shehab

Some reports were accepted as high with similar impact: https://solodit.xyz/issues/h-2-possible-dos-in-rollerperiphery-approve-function-sherlock-sense-sense-git I don't know if am missing something.

IAm0x52

While I acknowledge the similarity of the issues, I have made my escalation based on current Sherlock rules which overrides previous judgments. Loss of functionality is a medium issue. The DOS could be over a year but only for some ranges/LP tokens which is why it is also medium.

Ali-Shehab

But can't we send any token and make it DOS?

Czar102

As I understand, users wouldn't be able to create a borrow position for borrows with the active tick within the borrow tick range. This is a partial loss of the core functionality of the protocol, but doesn't lock any funds for >1 year. There also doesn't seem to be any loss of funds.

Current interpretation of the Sherlock rules imply that when there is no loss of funds or funds locked, it is not a high issue. I think it should be downgraded to medium, so will be accepting the escalation if there are not counterarguments.

Ali-Shehab

Also, I want to remind you that there is an issue that doesn't relate to this bug: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/37

Czar102

The first escalation here concerns another issue, #37.

Will be accepting both escalations. This will be downgraded to medium and #37 is not a duplicate. It is invalid.

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- ali-shehab: accepted
- IAm0x52: accepted

IAm0x52

Fix looks good. Underflow of this calculation is prevented via a ternary operator

Issue M-6: `computePoolAddress()` will not work on ZkSync Era

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/104

Found by

0x52, HHK, shogoki

When using the wagmi protocol, multiple swap can happen when borrowing or repaying a position. When the swap uses Uniswap v3 it checks that the callback is a pool by computing the address but the computation won't match on ZkSync Era.

Vulnerability Detail

When borrowing or repaying a position a user can either use a custom router that was approved by the wagmi team to make the swaps required or can use Uniswap v3 as a fallback.

When using the Uniswap v3 as a fallback the _v3SwapExactInput() internal function is being called. This function uses computePoolAddress() to find the pool address to use. computePoolAddress() is also used during the uniswapV3SwapCallback() to make sure the `msg.sender` is a valid pool.

On ZkSync Era the `create2` addresses are not computed the same way see here.

This will result in the swaps on Uniswapv3 to revert. If a user was able to open a position using a custom router but the custom router is removed later on by the team or if the liquidity was one sided so no swap happened. The borrower and liquidators could find themself not able to close the positions until a new router is whitelisted.

The borrower could be forced to pay collateral for a longer time as he won't be able to close his position.

Impact

Medium. Unlikely to happen but would result in short-term DOS and more fees paid by the borrower.

Code Snippet

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/b33752757fd6a9f404b8577c1eae6c5774b3a0db/wagmi-leverage/contracts/abstract/ApproveSwapAndPay.sol#L146 https://github.com/sherlock-audit/2023-10-real-wagmi/blob/b33752757fd6a9f404b8577c1eae6c5774b3a0db/wagmi-leverage/contracts/abstract/ApproveSwapAndPay.sol#L204 https://github.com/sherlock-audit/2023-10-real-wag

mi/blob/b33752757fd6a9f404b8577c1eae6c5774b3a0db/wagmi-leverage/contracts/abstract/ApproveSwapAndPay.sol#L271

Tool used

Manual Review

Recommendation

Consider calling the Uniswap factory getter `getPool()` to get the address of the pool.

Discussion

fann95

This is too obvious a problem with which this project simply will not work. We know about this, so we will make changes before deployment in ZkSync Era.

Czar102

As the contest readme states, watsons were to consider zkSync as one of the chains the code in scope was to be deployed on. If watsons couldn't have known that a modification of the code in scope would be deployed on zkSync, I don't see a reason to invalidate this issue, even if it was previously considered by the protocol team and/or is trivial.

fann95

I'm pasting the solution for Sherlock, but we don't plan to make any fixes to this issue right now.

Issue M-7: Wrong `accLoanRatePerSeconds` in `repay()` can lead to underflow

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/119

Found by

HHK, handsomegiraffe

When a Lender call the repay() function of the LiquidityBorrowingManager contract to do an emergency liquidity restoration using `isEmergency = true`, the `borrowingStorage.accLoanRatePerSeconds` is updated if the borrowing position hasn't been fully closed.

The computation is made in sort that the missing collateral can be computed again later so we don't loose the missing fees in case someone take over or the borrower decide to reimburse the fees.

But this computation is wrong and can lead to underflow.

Vulnerability Detail

Because the repay() function resets the `dailyRateCollateralBalance` to 0 when the lender call didn't fully close the position. We want to be able to compute the missing collateral again.

To do so we substract the percentage of collateral not paid to the `accLoanRatePerSeconds` so on the next call we will be adding extra second of fees that will allow the contract to compute the missing collateral.

The problem lies in the fact that we compute a percentage using the borrowed amount left instead of the initial borrow amount causing the percentage to be higher. In practice this do allows the contract to recompute the missing collateral.

But in the case of the missing `collateralBalance` or `removedAmt` being very high (ex: multiple days not paid or the loan removed was most of the position's liquidity) we might end up with a percentage higher than the `accLoanRatePerSeconds` which will cause an underflow.

In case of an underflow the call will revert and the lender will not be able to get his tokens back.

Consider this POC that can be copied and pasted in the test files (replace all tests and just keep the setup & NFT creation):

```
it("Updated accRate is incorrect", async () => {
        const amountWBTC = ethers.utils.parseUnits("0.05", 8); //token0
        let deadline = (await time.latest()) + 60;
```

```
        const minLeverageDesired = 50;
        const maxCollateralWBTC = amountWBTC.div(minLeverageDesired);

        const loans = [
            {
                liquidity: nftpos[3].liquidity,
                tokenId: nftpos[3].tokenId,
            },
            {
                liquidity: nftpos[5].liquidity,
                tokenId: nftpos[5].tokenId,
            },
        ];

        const swapParams: ApproveSwapAndPay.SwapParamsStruct = {
            swapTarget: constants.AddressZero,
            swapAmountInDataIndex: 0,
            maxGasForCall: 0,
            swapData: swapData,
        };

        const borrowParams = {
            internalSwapPoolfee: 500,
            saleToken: WETH_ADDRESS,
            holdToken: WBTC_ADDRESS,
            minHoldTokenOut: amountWBTC,
            maxCollateral: maxCollateralWBTC,
            externalSwap: swapParams,
            loans: loans,
        };

        //borrow tokens
        await borrowingManager.connect(bob).borrow(borrowParams, deadline);

        await time.increase(3600 * 72); //72h so 2 days of missing collateral
        deadline = (await time.latest()) + 60;

        const borrowingKey = await
↪   borrowingManager.userBorrowingKeys(bob.address, 0);

        let repayParams = {
            isEmergency: true,
            internalSwapPoolfee: 0,
            externalSwap: swapParams,
            borrowingKey: borrowingKey,
            swapSlippageBP1000: 0,
        };
```

```
        const oldBorrowingInfo = await
↪   borrowingManager.borrowingsInfo(borrowingKey);
        const dailyRateCollateral = await
↪   borrowingManager.checkDailyRateCollateral(borrowingKey);

        //Alice emergency repay but it reverts with 2 days of collateral missing
        await expect(borrowingManager.connect(alice).repay(repayParams,
↪   deadline)).to.be.revertedWithPanic();
    });
```

Impact

Medium. Lender might not be able to use `isEmergency` on `repay()` and will have to do a normal liquidation if he want his liquidity back.

Code Snippet

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/b33752757fd6a9f404b8577c1eae6c5774b3a0db/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L532 https://github.com/sherlock-audit/2023-10-real-wagmi/blob/b33752757fd6a9f404b8577c1eae6c5774b3a0db/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L613

Tool used

Manual Review

Recommendation

Consider that when a lender do an emergency liquidity restoration they give up on their collateral missing and so use the initial amount in the computation instead of borrowed amount left.

```
borrowingStorage.accLoanRatePerSeconds =
                    holdTokenRateInfo.accLoanRatePerSeconds -
                    FullMath.mulDiv(
                        uint256(-collateralBalance),
                        Constants.BP,
                        borrowing.borrowedAmount + removedAmt //old amount
                    );
```

Discussion

fann95

I think you are wrong in your conclusions. Try playing with emergency close tests. https://github.com/sherlock-audit/2023-10-real-wagmi/blob/b33752757fd6a9f404b8577c1eae6c5774b3a0db/wagmi-leverage/test/WagmiLeverageTests.ts#L990

The calculation must be made with the new borrowing.borrowedAmount, since in the future it will be used in the next calculation. The commission debt should be maintained and increase over time, and in your case, it is decreasing.

fann95

this is your option \

and this is my option \

In your version, the fee debt is reduced, although no one is paying it off.

HHK-ETH

Escalate

While using the new `borrowing.borrowedAmount` make sense to keep the missing collateral balance it will lead to underflow if the `holdTokenRateInfo.accLoanRatePerSeconds` is too low (ex: the first user to borrow so `accLoanRatePerSeconds` started at the same time).

To find the missing collateral balance with a smaller borrowed amount, the only solution is to increase the percentage. But like I said, if you increase the percentage you might end up underflowing as the `accLoanRatePerSeconds` could be smaller.

Here is a simple example:

- You have 10 tokens of collateral missing and 200 tokens total and we remove 50 tokens. The rate is 1% daily.
- 10 / 200 * 100 = 5% is the percentage value of the missing collateral compared to the amount borrowed.
- Since we were the first to borrow `accLoanRatePerSeconds` is pretty close to the missing collateral percentage, minus 1 day (user deposits at least 1 day of collateral when borrowing). 200*1/100 = 2 so the `accLoanRatePerSeconds` is 10+2 / 200 * 100 = 6%.
- In the contract what we do instead is 10 / (200-50) * 100 = 6.66%. We do this because we want to be able to find the missing collateral using a smaller balance during the next call as the state is updated.
- The substraction underflow because 6.66% > 6%.

I did play with the emergency close tests. Here I reused it and added comments:

```
it("emergency repay will be successful for PosManNFT owner if the collateral is
↪  depleted", async () => {
        let debt: LiquidityBorrowingManager.BorrowingInfoExtStructOutput[] =
```

```
            await borrowingManager.getBorrowerDebtsInfo(bob.address);

        await time.increase(debt[1].estimatedLifeTime.toNumber() + 3600 * 36);
↪   //add 36 hours as an example

        debt = await borrowingManager.getBorrowerDebtsInfo(bob.address);
↪   //update debt variable

        let borrowingKey = await borrowingManager.userBorrowingKeys(bob.address,
↪   1);

        let swap_params = ethers.utils.defaultAbiCoder.encode(
            ["address", "address", "uint256", "uint256"],
            [constants.AddressZero, constants.AddressZero, 0, 0]
        );
        swapData = swapIface.encodeFunctionData("swap", [swap_params]);

        let swapParams: ApproveSwapAndPay.SwapParamsStruct = {
            swapTarget: constants.AddressZero,
            swapAmountInDataIndex: 0,
            maxGasForCall: 0,
            swapData: swapData,
        };

        let params: LiquidityBorrowingManager.RepayParamsStruct = {
            isEmergency: true, //emergency
            internalSwapPoolfee: 0,
            externalSwap: swapParams,
            borrowingKey: borrowingKey,
            swapSlippageBP1000: 0,
        };

        //console.log(debt);
        let loans: LiquidityManager.LoanInfoStructOutput[] = await
↪   borrowingManager.getLoansInfo(borrowingKey);
        expect(loans.length).to.equal(3);

        let tokenRate = await borrowingManager.getHoldTokenDailyRateInfo(
            debt[1].info.saleToken,
            debt[1].info.holdToken
        );

        // We're going to repay alice loan which is [0] in loans array
        const aliceLoanPercentage = loans[0].liquidity
            .mul(100)

↪   .div(loans[0].liquidity.add(loans[1].liquidity).add(loans[2].liquidity));
```


```
        console.log("Alice % of total borrow: " + aliceLoanPercentage); //Alice
↪  liquidity is around 26% of the borrowing position

        //Here we do like the smart contract does, try to see how much
↪  accLoanRatePerSeconds we need to remove to find back our missing collateral
        //without removedAmount we use the new borrowed amount (current
↪  implementation)
        let accToRemoveWithoutRemovedAmount = debt[1].collateralBalance
            .mul(-1)
            .mul(10_000)
            .div(debt[1].info.borrowedAmount.mul(100 -
↪  aliceLoanPercentage.toNumber()).div(100));
        //with removedAmount (proposed solution), we forfeit alice part of
↪  missing collateral since she emergency withdraw
        let accToRemoveWithRemovedAmount = debt[1].collateralBalance
            .mul(-1)
            .mul(10_000)
            .div(debt[1].info.borrowedAmount);

        //this is the accLoanRatePerSeconds that we're going to substract from
        console.log("% to remove: " + accToRemoveWithoutRemovedAmount);
        console.log("accLoanRatePerSeconds: " +
↪  tokenRate.holdTokenRateInfo.accLoanRatePerSeconds);

        //for now even after 36hours we're still less than accLoanRatePerSeconds
↪  so no underflow
        console.log("% to remove is less than accLoanRatePerSeconds");
        expect(tokenRate.holdTokenRateInfo.accLoanRatePerSeconds.gt(accToRemoveW
↪  ithoutRemovedAmount));

        //let's add 12 more hours
        await time.increase(3600 * 12); //add 12 hours

        tokenRate = await
↪  borrowingManager.getHoldTokenDailyRateInfo(debt[1].info.saleToken,
↪  debt[1].info.holdToken); //update token rate
        debt = await borrowingManager.getBorrowerDebtsInfo(bob.address);
↪  //update debt variable
        accToRemoveWithoutRemovedAmount = debt[1].collateralBalance
            .mul(-1)
            .mul(10_000)
            .div(debt[1].info.borrowedAmount.mul(100 -
↪  aliceLoanPercentage.toNumber()).div(100));
        accToRemoveWithRemovedAmount = debt[1].collateralBalance.mul(-1).mul(10_
↪  000).div(debt[1].info.borrowedAmount);

        console.log("% to remove: " + accToRemoveWithoutRemovedAmount);
```

```
        console.log("accLoanRatePerSeconds: " +
↪  tokenRate.holdTokenRateInfo.accLoanRatePerSeconds);

        //the accToRemoveWithoutRemovedAmount is now greater than
↪  accLoanRatePerSeconds
        console.log("% to remove is greater than accLoanRatePerSeconds thus it
↪  will underflow");
        expect(tokenRate.holdTokenRateInfo.accLoanRatePerSeconds.lt(accToRemoveW
↪  ithoutRemovedAmount));

        let deadline = (await time.latest()) + 60;

        //this is going to revert with underflow
        await expect(borrowingManager.connect(alice).repay(params,
↪  deadline)).to.be.reverted;
        /**
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[0].tokenId)).to.be.equal(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[1].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[2].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[3].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[4].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[5].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getBorrowerDebtsCount(bob.address)).to.be.equal(2);

        debt = await borrowingManager.getBorrowerDebtsInfo(bob.address);
        //console.log(debt);
        loans = await borrowingManager.getLoansInfo(borrowingKey);
        expect(loans.length).to.equal(2);

        await time.increase(100);
        deadline = (await time.latest()) + 60;
        await expect(borrowingManager.connect(bob).repay(params, deadline))
            .to.emit(borrowingManager, "EmergencyLoanClosure")
            .withArgs(bob.address, bob.address, borrowingKey);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[0].tokenId)).to.be.equal(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[1].tokenId)).to.be.equal(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[2].tokenId)).to.be.gt(0);
```

```
        expect(await
↪   borrowingManager.getLenderCreditsCount(nftpos[3].tokenId)).to.be.gt(0);
        expect(await
↪   borrowingManager.getLenderCreditsCount(nftpos[4].tokenId)).to.be.gt(0);
        expect(await
↪   borrowingManager.getLenderCreditsCount(nftpos[5].tokenId)).to.be.gt(0);
        expect(await
↪   borrowingManager.getBorrowerDebtsCount(bob.address)).to.be.equal(2);
        debt = await borrowingManager.getBorrowerDebtsInfo(bob.address);
        //console.log(debt);
        loans = await borrowingManager.getLoansInfo(borrowingKey);
        expect(loans.length).to.equal(1);

        await time.increase(100);
        deadline = (await time.latest()) + 60;
        await expect(borrowingManager.connect(owner).repay(params, deadline))
            .to.emit(borrowingManager, "EmergencyLoanClosure")
            .withArgs(bob.address, owner.address, borrowingKey);
        expect(await
↪   borrowingManager.getLenderCreditsCount(nftpos[0].tokenId)).to.be.equal(0);
        expect(await
↪   borrowingManager.getLenderCreditsCount(nftpos[1].tokenId)).to.be.equal(0);
        expect(await
↪   borrowingManager.getLenderCreditsCount(nftpos[2].tokenId)).to.be.equal(0);
        expect(await
↪   borrowingManager.getLenderCreditsCount(nftpos[3].tokenId)).to.be.gt(0);
        expect(await
↪   borrowingManager.getLenderCreditsCount(nftpos[4].tokenId)).to.be.gt(0);
        expect(await
↪   borrowingManager.getLenderCreditsCount(nftpos[5].tokenId)).to.be.gt(0);
        expect(await
↪   borrowingManager.getBorrowerDebtsCount(bob.address)).to.be.equal(1);*/
    });
```

In this example Alice represents only 26% of the borrowed amount and the `accLoanRatePerSeconds` started before this loan so a longer time is needed (here I did 48hours) but if the user calling emergency repay has a much higher percentage or it's the first borrow it wouldn't need as much time.

I understand that my solution might not be what you want, but the underflow problem do exist.

sherlock-admin2

> Escalate
>
> While using the new `borrowing.borrowedAmount` make sense to keep the missing collateral balance it will lead to underflow if the

`holdTokenRateInfo.accLoanRatePerSeconds` is too low (ex: the first user to borrow so `accLoanRatePerSeconds` started at the same time).

To find the missing collateral balance with a smaller borrowed amount, the only solution is to increase the percentage. But like I said, if you increase the percentage you might end up underflowing as the `accLoanRatePerSeconds` didn't change.

Here is a simple example:

- You have 10 tokens of collateral missing and 200 tokens total and we remove 50 tokens. The rate is 1% daily.
- 10 / 200 * 100 = 5% is the percentage value of the missing collateral compared to the amount borrowed.
- Since we were the first to borrow `accLoanRatePerSeconds` is pretty close to the missing collateral percentage, minus 1 day (user deposits at least 1 day of collateral when borrowing). 200*1/100 = 2 so the `accLoanRatePerSeconds` is 10+2 / 200 * 100 = 6%.
- In the contract what we do instead is 10 / (200-50) * 100 = 6.66%. We do this because we want to be able to find the missing collateral using a smaller balance during the next call as the state is updated.
- The substraction underflow because 6% > 5%.

I did play with the emergency close tests. Here I reused it and added comments:

```
it("emergency repay will be successful for PosManNFT owner if the
↪  collateral is depleted", async () => {
        let debt: LiquidityBorrowingManager.BorrowingInfoExtStructOutput[] =
            await borrowingManager.getBorrowerDebtsInfo(bob.address);

        await time.increase(debt[1].estimatedLifeTime.toNumber() + 3600 *
↪  36); //add 36 hours as an example

        debt = await borrowingManager.getBorrowerDebtsInfo(bob.address);
↪  //update debt variable

        let borrowingKey = await
↪  borrowingManager.userBorrowingKeys(bob.address, 1);

        let swap_params = ethers.utils.defaultAbiCoder.encode(
            ["address", "address", "uint256", "uint256"],
            [constants.AddressZero, constants.AddressZero, 0, 0]
        );
        swapData = swapIface.encodeFunctionData("swap", [swap_params]);
```

```
        let swapParams: ApproveSwapAndPay.SwapParamsStruct = {
            swapTarget: constants.AddressZero,
            swapAmountInDataIndex: 0,
            maxGasForCall: 0,
            swapData: swapData,
        };

        let params: LiquidityBorrowingManager.RepayParamsStruct = {
            isEmergency: true, //emergency
            internalSwapPoolfee: 0,
            externalSwap: swapParams,
            borrowingKey: borrowingKey,
            swapSlippageBP1000: 0,
        };

        //console.log(debt);
        let loans: LiquidityManager.LoanInfoStructOutput[] = await
↪   borrowingManager.getLoansInfo(borrowingKey);
        expect(loans.length).to.equal(3);

        let tokenRate = await borrowingManager.getHoldTokenDailyRateInfo(
            debt[1].info.saleToken,
            debt[1].info.holdToken
        );

        // We're going to repay alice loan which is [0] in loans array
        const aliceLoanPercentage = loans[0].liquidity
            .mul(100)
            .div(loans[0].liquidity.add(loans[1].liquidity).add(loans[2].li
↪   quidity));
        console.log(aliceLoanPercentage); //Alice liquidity is around 26%
↪   of the borrowing position

        //Here we do like the smart contract does, try to see how much
↪   accLoanRatePerSeconds we need to remove to find back our missing
↪   collateral
        //without removedAmount (proposed solution), we forfeit alice part
↪   of missing collateral since she emergency withdraw
        let accToRemoveWithoutRemovedAmount = debt[1].collateralBalance
            .mul(-1)
            .mul(10_000)
            .div(debt[1].info.borrowedAmount.mul(100 -
↪   aliceLoanPercentage.toNumber()).div(100));
        //with removedAmount (current implmentation)
        let accToRemoveWithRemovedAmount = debt[1].collateralBalance
            .mul(-1)
            .mul(10_000)
            .div(debt[1].info.borrowedAmount);
```


```
        //this is the accLoanRatePerSeconds that we're going to substract
↪  from
        console.log(tokenRate.holdTokenRateInfo.accLoanRatePerSeconds);

        //for now even after 36hours we're still less than
↪  accLoanRatePerSeconds so no underflow
        expect(tokenRate.holdTokenRateInfo.accLoanRatePerSeconds.gt(accToRe
↪  moveWithRemovedAmount));

        //let's add 12 more hours
        await time.increase(debt[1].estimatedLifeTime.toNumber() + 3600 *
↪  12); //add 12 hours

        debt = await borrowingManager.getBorrowerDebtsInfo(bob.address);
↪  //update debt variable
        accToRemoveWithoutRemovedAmount = debt[1].collateralBalance
            .mul(-1)
            .mul(10_000)
            .div(debt[1].info.borrowedAmount.mul(100 -
↪  aliceLoanPercentage.toNumber()).div(100));
        accToRemoveWithRemovedAmount = debt[1].collateralBalance.mul(-1).mu
↪  l(10_000).div(debt[1].info.borrowedAmount);

        //the accToRemoveWithRemovedAmount is now greater than
↪  accLoanRatePerSeconds
        expect(tokenRate.holdTokenRateInfo.accLoanRatePerSeconds.lt(accToRe
↪  moveWithRemovedAmount));

        let deadline = (await time.latest()) + 60;

        //this is going to revert with underflow
        await expect(borrowingManager.connect(alice).repay(params,
↪  deadline))
            .to.emit(borrowingManager, "EmergencyLoanClosure")
            .withArgs(bob.address, alice.address, borrowingKey);

        expect(await borrowingManager.getLenderCreditsCount(nftpos[0].token
↪  Id)).to.be.equal(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[1].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[2].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[3].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[4].tokenId)).to.be.gt(0);
```


```
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[5].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getBorrowerDebtsCount(bob.address)).to.be.equal(2);

        debt = await borrowingManager.getBorrowerDebtsInfo(bob.address);
        //console.log(debt);
        loans = await borrowingManager.getLoansInfo(borrowingKey);
        expect(loans.length).to.equal(2);
        /**
        await time.increase(100);
        deadline = (await time.latest()) + 60;
        await expect(borrowingManager.connect(bob).repay(params, deadline))
            .to.emit(borrowingManager, "EmergencyLoanClosure")
            .withArgs(bob.address, bob.address, borrowingKey);
        expect(await borrowingManager.getLenderCreditsCount(nftpos[0].token
↪  Id)).to.be.equal(0);
        expect(await borrowingManager.getLenderCreditsCount(nftpos[1].token
↪  Id)).to.be.equal(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[2].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[3].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[4].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[5].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getBorrowerDebtsCount(bob.address)).to.be.equal(2);
        debt = await borrowingManager.getBorrowerDebtsInfo(bob.address);
        //console.log(debt);
        loans = await borrowingManager.getLoansInfo(borrowingKey);
        expect(loans.length).to.equal(1);

        await time.increase(100);
        deadline = (await time.latest()) + 60;
        await expect(borrowingManager.connect(owner).repay(params,
↪  deadline))
            .to.emit(borrowingManager, "EmergencyLoanClosure")
            .withArgs(bob.address, owner.address, borrowingKey);
        expect(await borrowingManager.getLenderCreditsCount(nftpos[0].token
↪  Id)).to.be.equal(0);
        expect(await borrowingManager.getLenderCreditsCount(nftpos[1].token
↪  Id)).to.be.equal(0);
        expect(await borrowingManager.getLenderCreditsCount(nftpos[2].token
↪  Id)).to.be.equal(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[3].tokenId)).to.be.gt(0);
```


```
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[4].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getLenderCreditsCount(nftpos[5].tokenId)).to.be.gt(0);
        expect(await
↪  borrowingManager.getBorrowerDebtsCount(bob.address)).to.be.equal(1);*/
    });
```

> In this example Alice represents only 26% of the borrowed amount and the `accLoanRatePerSeconds` started before this loan so a longer time is needed (here I did 48hours) but if the user calling emergency repay has a much higher percentage or it's the first borrow it wouldn't need as much time.
>
> I understand that my solution might not be what you want, but the underflow problem do exist.

The escalation could not be created because you are not exceeding the escalation threshold.

You can view the required number of additional valid issues/judging contest payouts in your Profile page, in the Sherlock webapp.

fann95

```
debt = await borrowingManager.getBorrowerDebtsInfo(bob.address); //update
↪  debt variable
```

tokenRate also needs to be updated like the debt variable

HHK-ETH

Right, my bad sorry wrote the test quickly. Updated my comment with the new token rate, `repay` still underflow.

HHK-ETH

@fann95 @cvetanovv sorry to bother but would appreciate if you could read the escalation comment again and run the 2 POCs provided (comment and initial finding) before escalation period ends.

The `repay()` reverts with underflow when increasing the missing collateral to a certain amount (just a few days in the examples given). I tried to explain that it comes from the `accLoanRatePerSeconds` calculation. I'll happily take the blame if I'm wrong and will be sorry for the time wasted but if I'm right this is problably something we want to fix or lenders might not be able to call `repay` and will have to count on liquidators.

chewonithard

issue 195 is also similar. I think both issues should be reconsidered to be valid.

This bug is easily replicated using protocol's own default test file WagmiLeverageTests.ts by changing line 1040 to 100_000 seconds (~1 day)

Expecting an under-collateralized loan to not be liquidated in ~1 day is not unrealistic, after which the failure of emergency mode represents a significant impact to a critical protocol function.

Czar102

@fann95 @cvetanovv could you take a look at this and #195?

cvetanovv

Should be duplicated and rather seem valid to me

Czar102

@fann95 could you post your opinion on this issue? It seems to be valid.

fann95

I looked at this problem one more and realized: yes, this is a mistake. When a liquidity provider withdraws its liquidity, it also surrenders its fee debt, so the debt must be reduced in proportion to the reduction in liquidity.

fann95

I came to an interesting conclusion: the borrowingStorage.accLoanRatePerSeconds does not need to be recalculated. It should remain the same as it was, and the borrowed amount should be decreased only. So, accordingly, the fee debt will be reduced during a future recalculation. Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/4d355d8cafc86f1341af7b5acfb485a2a33ce61a

Czar102

@fann95 may I ask why do you consider this a high severity issue? It seems the watson classified this as medium severity.

Czar102

Agree with medium severity. Will be accepting the escalation.

IAm0x52

Fix looks good. `borrowingStorage.accLoanRatePerSeconds` does not need to be recalculated and so the calculation has been removed.

Issue M-8: Borrower collateral that they are owed can get stuck in Vault and not sent back to them after calling `repay`

Source: https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/122

Found by

HHK, detectiveking

There's a case where a borrower calls `borrow`, perhaps does a bunch of intermediate actions like calling `increaseCollateralBalance`, and then calls `repay` a short while later (so fees haven't had a time to increase), but the collateral they are owed is stuck in the `Vault` instead of being sent back to them after they repay.

Vulnerability Detail

First, let's say that a borrower called `borrow` in `LiquidityBorrowingManager`. Then, they call increase `increaseCollateralBalance` with a large collateral amount. A short time later, they decide they want to repay so they call `repay`.

In `repay`, we have the following code:

```
if (
    collateralBalance > 0 &&
    (currentFees + borrowing.feesOwed) / Constants.COLLATERAL_BALANCE_PRECISION >
    Constants.MINIMUM_AMOUNT
) {
    liquidationBonus +=
        uint256(collateralBalance) /
        Constants.COLLATERAL_BALANCE_PRECISION;
} else {
    currentFees = borrowing.dailyRateCollateralBalance;
}
```

Notice that if we have `collateralBalance > 0` BUT `!((currentFees + borrowing.feesOwed) / Constants.COLLATERAL_BALANCE_PRECISION > Constants.MINIMUM_AMOUNT)` (i.e. the first part of the if condition is fine but the second is not. It makes sense the second part is not fine because the borrower is repaying not long after they borrowed, so fees haven't had a long time to accumulate), then we will still go to `currentFees = borrowing.dailyRateCollateralBalance;` but we will not do:

```
liquidationBonus +=
    uint256(collateralBalance) /
```

```
    Constants.COLLATERAL_BALANCE_PRECISION;
```

However, later on in the code, we have:

```
Vault(VAULT_ADDRESS).transferToken(
    borrowing.holdToken,
    address(this),
    borrowing.borrowedAmount + liquidationBonus
);
```

So, the borrower's collateral will actually not even be sent back to the LiquidityBorrowingManager from the Vault (since we never incremented `liquidationBonus`). We later do:

```
_pay(borrowing.holdToken, address(this), msg.sender, holdTokenBalance);
_pay(borrowing.saleToken, address(this), msg.sender, saleTokenBalance);
```

So clearly the user will not receive their collateral back.

Impact

User's collateral will be stuck in Vault when it should be sent back to them. This could be a large amount of funds if for example `increaseCollateralBalance` is called first.

Code Snippet

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L565-L575

https://github.com/sherlock-audit/2023-10-real-wagmi/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L632-L670

Tool used

Manual Review

Recommendation

You should separate:

```
if (
    collateralBalance > 0 &&
    (currentFees + borrowing.feesOwed) / Constants.COLLATERAL_BALANCE_PRECISION >
    Constants.MINIMUM_AMOUNT
) {
```

```
    liquidationBonus +=
        uint256(collateralBalance) /
        Constants.COLLATERAL_BALANCE_PRECISION;
} else {
    currentFees = borrowing.dailyRateCollateralBalance;
}
```

Into two separate if statements. One should check if `collateralBalance > 0`, and if so, increment liquidationBonus. The other should check `(currentFees + borrowing.feesOwed) / Constants.COLLATERAL_BALANCE_PRECISION > Constants.MINIMUM_AMOUNT` and if not, set `currentFees = borrowing.dailyRateCollateralBalance;`.

Discussion

fann95

Fixed: https://github.com/RealWagmi/wagmi-leverage/commit/96ad6c13b3f5e37e3b13fd53991f69c5d1b07f87

IAm0x52

Escalate

This is a valid issue but it should be medium rather than high as it only occurs under very specific circumstances. For most collaterals this would only occur of the user borrows and repays in the same block or if the borrow amount is exceptionally low.

sherlock-admin2

> Escalate
>
> This is a valid issue but it should be medium rather than high as it only occurs under very specific circumstances. For most collaterals this would only occur of the user borrows and repays in the same block or if the borrow amount is exceptionally low.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

detectiveking123

Disagree with the escalation.

To name a few things,

1. User could lose a large amount of funds if `increaseCollateralBalance` is called first, as my report mentions. The amount of funds lost here could be quite high.
2. The escalation says: "For most collaterals this would only occur of the user borrows and repays in the same block or if the borrow amount is exceptionally low." <- This seems incorrect, you could easily wait a few minutes before returning on a decently sized collateral and still lose some funds. But for sure the worst case is waiting a few seconds to return on a higher amount of funds. I argue that this is actually quite a common use case. This brings me to my next point.
3. Remember that this is a trading protocol at the end of the day. Borrowing and repaying is a trade. Many traders employ high frequency strategies, and might open a position and attempt to close a short time afterwards (even in the next block, for example). This is extremely common if you look at well known on-chain perp protocols. I maintain that a large number of traders would run into this issue and lose funds.

IAm0x52

A single 18 dp token is 10^18. At the minimum interest of 0.1% per day the interest per second is:

10^18 * 0.001 / 86400 = 11.574 * 10^9

This is such higher than the MINIMUM_AMOUNT of 1000. This means that any 18 dp token (unless each token was valued at $11,500,000 which currently doesn't exist) would only suffer from this if closed in the same block.

For 6 dp tokens:

10^6 * 0.001 / 86400 = 1.1574 * 10^-2

This means that 6 dp positions over 86400 would not suffer from this unless closed in the same block.

For ETH with a block time of 12 seconds, a position of 7200 could be closed by the next block and not suffer from this.

As stated above this only affects limited number of tokens/positions and should therefore be medium not high.

detectiveking123

For the dp point, I think USDC and USDT are tokens that constitute an extremely large portion of the market share here, so I disagree with your "limited number of tokens" point. Also, as you kind of alluded to, each (10^6) token can be worth more than $1.

Nonetheless, I personally think $86,400 per position is a lot of money (and the total number of funds at risk is much higher, since this bug can be repeated for every

position. This isn't even taking a potential call to `increaseCollateralBalance` into account.). Much faster chains, like FTM, also exist in scope, where block times are much lower and the probability of a new block coming out in one second or less after the previous is nontrivial.

The bigger point is that, even if `increaseCollateralBalance` is not called, this bug completely bricks the HFT use case. If you look at the top perp protocols, a pretty high percentage of volume is just bots flipping around futures in short time frames. I don't think a use case that many institutional investors rely on and probably constitutes over 50% of volume on existing top perp protocols should be considered "limited".

Here is Sherlock's criteria for High validity:

```
High: This vulnerability would result in a material loss of funds, and the cost of the attack is low (relative to the amount of funds lost). The attack path is possible with reasonable assumptions that mimic on-chain conditions. The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.
```

"This vulnerability would result in a material loss of funds, and the cost of the attack is low (relative to the amount of funds lost)" -> There is indeed a material loss of funds.

"The attack path is possible with reasonable assumptions that mimic on-chain conditions." -> Yes, the HFT use case is pretty reasonable and common. The HFT use case is completely bricked and will lead to a material loss of funds. I also think that starting off with a lower amount of collateral, calling increase collateral balance (you don't want to be afraid of liquidation), and then attempting to close your position a few minutes later (e.g. if price moves against you and you change your mind), is a reasonable use case that will lead to a large material loss of funds.

"The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team." -> Sponsor has confirmed that this is not an acceptable risk.

Czar102

It seems that the crux of the escalation argument is that this bug affects only a subset of use cases. I think the use cases affected are not "theoretical" enough not to deserve a high severity, they seem sensible. Even repaying in the same block should be allowed, for example could be used by a protocol utilizing this system that allows everyone to contribute.

The protocol team doesn't seem to have implemented considered `if` clause because of the formal correctness, rather real use cases.

Also, I don't follow this:

> For 6 dp tokens:

10^6 * 0.001 / 86400 = 1.1574 * 10^-2

This means that 6 dp positions over 86400 would not suffer from this unless closed in the same block.

It seems that the interest from an Ethereum block would be below 1 wei, certainly below `Constants.MINIMUM_AMOUNT`. But there is a good chance I am misunderstanding something.

@IAm0x52 please let me know if my approach makes sense.

IAm0x52

My argument stems from this set of judging rules which states that the issue is medium if it requires certain states/external conditions which it does. Also this isn't an attack. No one can repay or close your loan for you (besides liquidation which is irrelevant here). You would have to cause this loss for yourself.

By 86400 above I mean 86400 full 6 dp tokens i.e. 86400e6

Czar102

I am aware of Sherlock's rules. The conditionality in this issue is not of a state or external factors, but the use case of the protocol. In other words, this bug is not triggered if external conditions are met, or a specific state is achieved, which may be a reason to decrease severity to medium. As you said, triggering of this bug only depends on the use case of the protocol.

By executing actions that should normally work (and those actions have real, reasonable use cases), the user irreversibly loses funds which are locked in the contract forever.

I am planning to reject the escalation and leave this a high severity issue.

IAm0x52

I see. This seems to be a much different interpretation of Sherlock's rule than has been previously used. My comments have been based on how I have seen this judged previously. We are all on the same page it seems from the technical side of if/when this issue happens. If @Evert0x agrees with this interpretation of the high risk criteria based on potentially reasonable use cases then there is nothing more to talk about here. If this is the case, I would also appreciate a change to judging rules to reflect this new line of logic.

I personally don't see opening/adding collateral and closing in the exact same block as a common use case. I know HFT was mentioned above but since a single block is one specific point in time the only utility I see for that type of action would be arbitrage within the block. Given the large number of other more efficient ways to accomplish this it seems unlikely. Outside of single block closure the amount of time required between open and close to avoid this is a very short time (less than a

minute for all but the smallest 6 dp token positions) I also find it highly unlikely that this will occur often at all.

detectiveking123

"Also this isn't an attack. No one can repay or close your loan for you (besides liquidation which is irrelevant here). You would have to cause this loss for yourself."

Historically many issues that aren't an attack, but cause a loss of funds, have been classified as high.

https://github.com/sherlock-audit/2023-08-cooler-judging/issues/119 is one such example

Czar102

Discussed this issue with @Evert0x. The high severity was designed to distinguish issues that would *very likely cause protocol failure* if in production. This bug is triggered only in scenarios that are somewhat extreme (execution in the same block, e.g. after gas price normalizes while the frontend allows that, some very specific protocol integrations, etc.)

The use cases presented are potentially reasonable, but very specific. Decided to consider this a medium severity issue. **Escalation will be accepted and severity will be changed to medium.**

detectiveking123

It's not exactly execution in the same block though -- you can execute multiple blocks later and still lose a decent amount of money (e.g. thousands at stake 10 blocks later). HFT is also a use case that will constitute a large amount, if not majority, of the volume on the platform.

Will respect the outcome though, it is probably a borderline case.

Czar102

@detectiveking123 could you work out an example of a large loss in the longest timeframe possible? (most likely show payoff parameters for a worst case scenario)

Evert0x

@detectiveking123 will accept escalation and make Medium in 24 hours unless requested argument is provided.

HHK-ETH

> @detectiveking123 could you work out an example of a large loss in the longest timeframe possible? (most likely show payoff parameters for a worst case scenario)

Because `MINIMUM_AMOUNT` doesn't scale with the token's decimals like I mentionned in this issue, if you were to use a token with even lower decimals than 1e6 you

could end up with a much longer time frame.

If you want to long GUSD (2 decimals) against USDC (360k+ TVL on Uniswap v3), you can open a position with 5k GUSD with 0.1% fee and close it a day later and still be under `MINIMUM_AMOUNT` thus loosing your collateral balance.

5k GUSD -> 500000 wei `MINIMUM_AMOUNT` -> 1000

500000 * 0.1 / 100 = 500 < `MINIMUM_AMOUNT`

Amount lost will be very small most of the time as it will be the collateral for a day so between 0.05% and 1% according to current protocol's parameters.

Exception is if the user called `increaseCollateralBalance()` before changing his mind and closing the position in which case it can be way higher. An example using the case above is a user betting on a USDC depeg and so open a position to short it and long GUSD and put collateral for a few days by calling `increaseCollateralBalance()`, then a news comes in that makes the depeg unlikely anymore and so our user close his positions but looses his whole collateral.

So I guess the question is how likely is this kind of scenario to know if it falls in "*Causes a loss of funds but requires certain external conditions or specific states*".

Czar102

> Amount lost will be very small most of the time as it will be the collateral for a day so between 0.05% and 1% according to current protocol's parameters.

From my understanding, it was a loss of the whole collateral. So in the above case, it's $5k. Can you confirm @HHK-ETH @detectiveking123?

HHK-ETH

> > Amount lost will be very small most of the time as it will be the collateral for a day so between 0.05% and 1% according to current protocol's parameters.
>
> From my understanding, it was a loss of the whole collateral. So in the above case, it's $5k. Can you confirm @HHK-ETH @detectiveking123?

Yes it's a loss of the whole collateral but no collateral is only a percentage of the borrowing position since you don't control the tokens borrowed this protocol doesn't need you to have an overcollaterized position.

HHK-ETH

Updated the calculation in https://github.com/sherlock-audit/2023-10-real-wagmi-judging/issues/122#issuecomment-1807427259 as I made an error yesterday. New example is a day long.

Evert0x

Based on my interpretation this is a clear case for Medium severity. But if I understand correctly, you are advocating for high severity with your comment? @HHK-ETH

- Loss is between 0.05% and 1%
- Requires 2 decimal tokens (GUSD)
- User needs to call `increaseCollateralBalance()` before to increase the loss %

  So I guess the question is how likely is this kind of scenario to know if it falls in "Causes a loss of funds but requires certain external conditions or specific states".

You are also quoting the "How to identify a medium issue" here, so I'm kind of confused what your comment is intending to clarify. As medium was agreed upon earlier.

HHK-ETH

I was mostly trying to give extra examples that could help judges define the severity.

Although I have a duplicate of this finding and would benefit from it accepted as high, it seems that it fits better as a medium severity indeed.

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- IAm0x52: accepted

IAm0x52

Fix looks good. Minimum calculations have been moved to `_calcFeeCompensationUpToMin` which charges the user the minimum fee if the currently accrued fee is lower than the minimum.