SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Unstoppable
Prepared by: Sherlock
Lead Security Expert: stopthecap
Dates Audited: June 28 - July 5, 2023
Prepared on: August 16, 2023

Introduction

Democratizing access to DeFi with simple UI/UX and a suite of amazing products. More than a #DEX. On a mission to make centralized exchanges obsolete.

Scope

Repository: Unstoppable-DeFi/unstoppable-dex-audit

Branch: main

Commit: 4153c3e67ccc080032ba0bbaffd9a0c56a573070

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
9	8

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

stopthecap
0xyPhilic
kutugu
0x4non
0xStalin
pengun
0xDjango
Yuki
GimelSec
0x00ffDa
0xbepresent
0xpinky

Tricko
Dug
TheNaubit
ontheway

n33k
twicek
whiteh4t9527
Lalanda

chainNue
BugBusters
Bauer

Issue H-1: Wrong accounting of the storage balances results for the protocol to be in debt even when the bad debt is repaid.

Source:
https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/68

Found by

Yuki

Summary

Wrong accounting of the storage balances results for the protocol to be in debt even when the bad debt is repaid.

Vulnerability Detail

When a position is fully closed, it can result to accruing bad debt or being in profit and repaying the borrowed liquidity which all depends on the amount_out_received from the swap.

In a bad debt scenario the function calculates the bad debt and accrues it based on the difference between the position debt and the amount out received from the swap. And after that repays the liquidity providers with the same received amount from the swap.

The mapping total_debt_amount holds all debt borrowed from users, and this amount accrues interest with the time.

Prior to closing a position, the bad debt is calculated and accrued to the mapping bad_debt, but it isn't subtracted from the mapping total_debt_amount which holds all the debt even the bad one accrued through the time.

As the bad debt isn't subtracted from the total_debt_amount when closing a position, even after repaying the bad debt. It will still be in the total_debt_amount, which will prevent the full withdrawing of the liquidity funds.

Impact

The issue leads to liquidity providers unable to withdraw their full amount of funds, as even after repaying the bad debt it will still be in the total_debt_amount mapping.

Code Snippet

https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L233

Tool used

Manual Review

Recommendation

The only way to fix this problem is to repay the position debt amount prior to closing a position and not only the received amount from the swap. Because total_debt_amount holds the bad debt as well.

Discussion

CodingNameKiki

Escalate

This issue should be high, as it results to permanent loss of funds for liquidity providers.

What the issue leads to:

1. Double counting of the bad debt in the system, as it isn't subtracted from the mapping `total_debt_amount`.
2. When the bad debt is repaid, the system will still be in debt.
3. Liquidity providers will not be able to withdraw their full amount of funds, as a result the funds will be stuck in the system.

The referred issue describes the outcome of the issue well, but I am still going to provide an additional explanation in this escalation.

Detailed explanation

Part 1 - Showing how bad debt accrues.

The internal function `_close_position` is called when a position is fully closed.

There could be two scenarios when the function is called.

- `if amount_out_received >= position_debt_amount:` - the amount out received from the swap is bigger than the position debt, which indicates that the user is in profit and the whole debt amount is repaid and subtracted from the mapping total_debt_amount.

- `else` indicates that the amount out received from the swap is smaller than the position debt. As a result bad debt is accrued in the mapping bad_debt.

Conclusion:

- When bad debt is accrued, the system adds the amount of debt to the mapping bad_debt, but doesn't subtract the same amount from the mapping total_debt_amount.
- The mapping total_debt_amount holds all the debt borrowed from users and the extra amount of debt accrued through the time in terms of interest.
- As a result the bad debt is double counted in the system.

The system provides a way for users to repay the bad debt accrued in the mapping bad_debt.

- However as double counted, once repaid from the mapping bad_debt, the other same half will still persist in the mapping total_debt_amount.

Part 2 - Liquidity providers not able to withdraw their full amount of funds

So far we've explained the main part of the issue, but lets see the impact on the liquidity providers.

- When the function `withdraw_liquidity` is called, it checks that the system has the available amount of liquidity in order to perform the withdraw.

What we know so far:

- When the system accrues bad debt, it forgets to remove the particular amount of bad debt from the mapping total_debt_amount, which holds all the debt accrued through the time in terms of interest (debt + extra debt), as a result after bad debt is accrued the same amount of bad debt persist both in the mappings bad_debt and total_debt_amount.

What happens when calculating the available liquidity:

1. First we can see that the total liquidity amount is calculated based on the provided liquidity from the liquidity providers. Then the bad debt which the system holds is subtracted from it.
2. After that the system subtracts the total_debt_amount from the total liquidity.
3. As a result the same amount of bad debt accrued is subtracted twice from the total liquidity as it persist both in the mappings bad_debt and total_debt_amount.

Conclusion:

- The conclusion is that even if the bad debt is repaid from the mapping bad_debt, it will still persist in total_debt_amount.
- This amount of bad debt can't be removed from the mapping total_debt_amount and will be accrued in terms of interest through time and persist in it forever.
- Duo to this issue the liquidity providers will not be able to withdraw the full amount of funds they deposited, and the same amount of funds will be permanently locked in the system.

sherlock-admin2

Escalate

This issue should be high, as it results to permanent loss of funds for liquidity providers.

What the issue leads to:

1. Double counting of the bad debt in the system, as it isn't subtracted from the mapping `total_debt_amount`.
2. When the bad debt is repaid, the system will still be in debt.
3. Liquidity providers will not be able to withdraw their full amount of funds, as a result the funds will be stuck in the system.

The referred issue describes the outcome of the issue well, but I am still going to provide an additional explanation in this escalation.

Detailed explanation

Part 1 - Showing how bad debt accrues.

The internal function `_close_position` is called when a position is fully closed.

There could be two scenarios when the function is called.

- `if amount_out_received >= position_debt_amount:` - the amount out received from the swap is bigger than the position debt, which indicates that the user is in profit and the whole debt amount is repaid and subtracted from the mapping total_debt_amount.
- `else` indicates that the amount out received from the swap is smaller than the position debt. As a result bad debt is accrued in the mapping bad_debt.

Conclusion:

- When bad debt is accrued, the system adds the amount of debt to the mapping bad_debt, but doesn't subtract the same amount from the mapping total_debt_amount.
- The mapping total_debt_amount holds all the debt borrowed from users and the extra amount of debt accrued through the time in terms of interest.
- As a result the bad debt is double counted in the system.

The system provides a way for users to repay the bad debt accrued in the mapping bad_debt.

- However as double counted, once repaid from the mapping bad_debt, the other same half will still persist in the mapping total_debt_amount.

Part 2 - Liquidity providers not able to withdraw their full amount of funds

So far we've explained the main part of the issue, but lets see the impact on the liquidity providers.

- When the function `withdraw_liquidity` is called, it checks that the system has the available amount of liquidity in order to perform the withdraw.

What we know so far:

- When the system accrues bad debt, it forgets to remove the particular amount of bad debt from the mapping total_debt_amount, which holds all the debt accrued through the time in terms of interest (debt + extra debt), as a result after bad debt is accrued the same amount of bad debt persist both in the mappings bad_debt and total_debt_amount.

What happens when calculating the available liquidity:

1. First we can see that the total liquidity amount is calculated based on the provided liquidity from the liquidity providers. Then the bad debt which the system holds is subtracted from it.
2. After that the system subtracts the total_debt_amount from the total liquidity.
3. As a result the same amount of bad debt accrued is subtracted twice from the total liquidity as it persist both in the mappings bad_debt and total_debt_amount.

Conclusion:

- The conclusion is that even if the bad debt is repaid from the mapping bad_debt, it will still persist in total_debt_amount.
- This amount of bad debt can't be removed from the mapping total_debt_amount and will be accrued in terms of interest through time and persist in it forever.
- Duo to this issue the liquidity providers will not be able to withdraw the full amount of funds they deposited, and the same amount of funds will be permanently locked in the system.

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.

SilentYuki

The escalation is correct, the issue should be high severity.

141345

Recommendation: change the original judging, to high severity.

> Escalate
>
> This issue should be high, as it results to permanent loss of funds for liquidity providers.
>
> What the issue leads to:
>
> 1. Double counting of the bad debt in the system, as it isn't subtracted from the mapping `total_debt_amount`.
> 2. When the bad debt is repaid, the system will still be in debt.
> 3. Liquidity providers will not be able to withdraw their full amount of funds, as a result the funds will be stuck in the system.
>
> The referred issue describes the outcome of the issue well, but I am still going to provide an additional explanation in this escalation.

Detailed explanation

Part 1 - Showing how bad debt accrues.

The internal function `_close_position` is called when a position is fully closed.

There could be two scenarios when the function is called.

- `if amount_out_received >= position_debt_amount:` - the amount out received from the swap is bigger than the position debt, which indicates that the user is in profit and the whole debt amount is repaid and subtracted from the mapping total_debt_amount.
- `else` indicates that the amount out received from the swap is smaller than the position debt. As a result bad debt is accrued in the mapping bad_debt.

Conclusion:

- When bad debt is accrued, the system adds the amount of debt to the mapping bad_debt, but doesn't subtract the same amount from the mapping total_debt_amount.
- The mapping total_debt_amount holds all the debt borrowed from users and the extra amount of debt accrued through the time in terms of interest.
- As a result the bad debt is double counted in the system.

The system provides a way for users to repay the bad debt accrued in the mapping bad_debt.

- However as double counted, once repaid from the mapping bad_debt, the other same half will still persist in the mapping total_debt_amount.

Part 2 - Liquidity providers not able to withdraw their full amount of funds

So far we've explained the main part of the issue, but lets see the impact on the liquidity providers.

- When the function `withdraw_liquidity` is called, it checks that the system has the available amount of liquidity in order to perform the withdraw.

What we know so far:

- When the system accrues bad debt, it forgets to remove the particular amount of bad debt from the mapping total_debt_amount, which holds all the debt accrued through the time in terms of interest (debt + extra debt), as a result after bad debt is accrued the same amount of bad debt persist both in the mappings bad_debt and total_debt_amount.

What happens when calculating the available liquidity:

1. First we can see that the total liquidity amount is calculated based on the provided liquidity from the liquidity providers. Then the bad debt which the system holds is subtracted from it.
2. After that the system subtracts the total_debt_amount from the total liquidity.
3. As a result the same amount of bad debt accrued is subtracted twice from the total liquidity as it persist both in the mappings bad_debt and total_debt_amount.

Conclusion:

- The conclusion is that even if the bad debt is repaid from the mapping bad_debt, it will still persist in total_debt_amount.
- This amount of bad debt can't be removed from the mapping total_debt_amount and will be accrued in terms of interest through time and persist in it forever.
- Duo to this issue the liquidity providers will not be able to withdraw the full amount of funds they deposited, and the same amount of funds will be permanently locked in the system.

The missed updated total debt will:

- continuing accruing interest
- reduce the available liquidity to use, so reduce income from normal operation
- gradually lock LP's fund

Based on the above, the long term impact could be deemed as "material loss of funds", high is appropriate.

hrishibhat

Result: High Unique Agree with Lead judge's comment. Considering this issue a high

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- CodingNameKiki: accepted

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/19

maarcweiss

Fixed by moving the `repay()` position logic to before the position is closed

Issue H-2: `reduce_margin_by_amount` in `Vault.reduce_position` is wrongly calculated

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/85

Found by

GimelSec

Summary

In `Vault.reduce_position`, some position tokens are sold and debt and margin are reduced. The remaining amount of debt and margin should maintain the same leverage as before. However, the reduced amount is wrongly calculated and as a result, the leverage is changed.

Vulnerability Detail

The calculation of reduced amount is implemented in `Vault.reduce_position`. https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L314 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L322 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L324

Suppose that the debt amount is 9000 USDC and the margin amount is also 9000 USDC. And the amount_out_received is 9000 USDC. Let's take a look of the calculation:

We can find out that the reduced amount of margin is 9000 and the reduced amount of debt is 0. The leverage is changed.

I also wrote a test to demonstrate the issue. (The test case is different from the previous case)

```
def test_reduce_position(vault, owner, weth, usdc, mock_router, eth_usd_oracle):
    eth_usd_oracle.set_answer(9000_00000000)
    assert vault.swap_router() == mock_router.address

    uid, amount_bought = vault.open_position(
        owner,  # account
        weth,  # position_token
        2 * 10**18,  # min_position_amount_out
        usdc,  # debt_token
        9000 * 10**6,  # debt_amount
        9000 * 10**6,  # margin_amount
```

```
    )

    position_before = vault.positions(uid)
    position_amount_before = position_before[6]
    assert position_amount_before == 2 * 10**18

    margin_amount_before = position_before[3]
    assert margin_amount_before == 9000 * 10**6

    debt_amount_before = vault.debt(uid)
    assert debt_amount_before == 9000 * 10**6

    vault.reduce_position(uid, int(position_amount_before/3), 6000 * 10**6)

    position_after = vault.positions(uid)

    debt_amount_after = vault.debt_shares_to_amount(usdc.address,
    ↪  position_after[4])
    print(debt_amount_after)
    assert debt_amount_after == pytest.approx(6000 * 10**6, abs = 10000000) //
    ↪  @audit it would fail since debt_amount_after  is 9000 * 10**6

    margin_amount_after = position_after[3]
    print(margin_amount_after)
    assert margin_amount_after == pytest.approx(6000 * 10**6, abs = 1000000)

    position_amount_after = position_after[6]
```

Besides, I want to mention that the usage of `pytest.approx` is incorrect in `test_reduce_position`. https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/tests/vault/test_reduce_position.py#L56

```
assert debt_amount_after == pytest.approx(6000 * 10**6, 10000000)
```

If the approximation means `6000 * 10 ** 6 ± 10000000`, it should be `pytest.approx(6000 * 10**6, abs=10000000)`. On the other hand, `pytest.approx(6000 * 10**6, 10000000)` considers numbers within a relative tolerance of the expected value. The approximation `6000 * 10 ** 6 ± (6000 * 10**6 * 10000000)` has too large of a range of values.

Impact

The reduced amounts of debt and margin are wrongly calculated. The leverage is changed after `reduce_position`.

Code Snippet

https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L314 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L322 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L324

Tool used

Manual Review

Recommendation

The correct calculation should be:

```
def reduce_position(
    _position_uid: bytes32, _reduce_by_amount: uint256, _min_amount_out: uint256
) -> uint256:
    ...
-   margin_debt_ratio: uint256 = position.margin_amount * PRECISION / debt_amount
+   margin_debt_ratio: uint256 = position.margin_amount * PRECISION / (
 ↪  debt_amount + position.margin_amount)

    amount_out_received: uint256 = self._swap(
        position.position_token, position.debt_token, _reduce_by_amount,
 ↪  min_amount_out
    )

    # reduce margin and debt, keep leverage as before
    reduce_margin_by_amount: uint256 = (
        amount_out_received * margin_debt_ratio / PRECISION
    )
    reduce_debt_by_amount: uint256 = amount_out_received -
 ↪  reduce_margin_by_amount

    position.margin_amount -= reduce_margin_by_amount

    burnt_debt_shares: uint256 = self._repay(position.debt_token,
 ↪  reduce_debt_by_amount)
    position.debt_shares -= burnt_debt_shares
    position.position_amount -= _reduce_by_amount

    self.positions[_position_uid] = position

    log PositionReduced(position.account, _position_uid, position,
 ↪  amount_out_received)
```

```
    return amount_out_received
```

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/3

maarcweiss

Fixed by using `debt_amount + position.margin_amount` instead of `debt_amount`

Issue H-3: Vault: The attacker can sandwich attack himself on swaps in open_position, close_position and reduce_position to make a bad debt

Source:
https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/140

Found by

0xyPhilic, Tricko, n33k, ontheway, pengun, stopthecap, whiteh4t9527

Summary

The attacker could call MarginDex to open/close postions in Vault. The swap slippage is controlled by the attacker. He can set the slippage to infinite to steal from swaps and leave a bad postion/debt.

Vulnerability Detail

In open_position of Vault, max_leverage is first checked to ensure the position that is about to open will be healthy. Then token is borrowed and swapped into postion token. The attacker can set slippage to infinite and sandwich attack the swap to steal almost all of the swapped token. Then the amount swapped out is recorded in postion and the postion will be undercollateralized.

Swaps in close_position and reduce_position are also vulnerable to this sandwich attack. The attacker can attack to steal from swap and make a bad debt.

Impact

The protocol could be drained because of this sandwich attack.

Code Snippet

https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L182-L184

https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L252-L257

Tool used

Manual Review

Recommendation

Call `_is_liquidatable` in the end of open_position and reduce_position.

Check pool price deviation to oracle price inside close_position.

Discussion

twicek

Escalate for 10 USDC The pools used in the margin dex are whitelisted and therefore highly liquid. Using a flashloan in such a pool to manipulate the reserves would not be economically viable since the flashloans are attached with a fee. It would not even be possible to repay the loan in the scenario described in the report (it would revert).

This issue should be at most medium.

PS: Note that it is not possible to sandwich other's transactions on Arbitrum. So the above assumes that a malicious user wants to sandwich its own transaction by making several function call in one transaction. I'm no MEV expert, so correct me if I'm wrong, but Arbitrum uses a sequencer to order transactions and does not have a mempool. Instead, transactions are ordered on a first come, first served basis. Therefore, it is only possible to back-run transactions. On Arbitrum MEV bots are competing for the lowest latency to the sequencer in order to identify profitable transactions and initiate a back-run.

sherlock-admin2

> Escalate for 10 USDC The pools used in the margin dex are whitelisted and therefore highly liquid. Using a flashloan in such a pool to manipulate the reserves would not be economically viable since the flashloans are attached with a fee. It would not even be possible to repay the loan in the scenario described in the report (it would revert).
>
> This issue should be at most medium.
>
> PS: Note that it is not possible to sandwich other's transactions on Arbitrum. So the above assumes that a malicious user wants to sandwich its own transaction by making several function call in one transaction. I'm no MEV expert, so correct me if I'm wrong, but Arbitrum uses a sequencer to order transactions and does not have a mempool. Instead, transactions are ordered on a first come, first served basis. Therefore, it is only possible to back-run transactions. On Arbitrum MEV bots are competing for the lowest latency to the sequencer in order to identify profitable transactions and initiate a back-run.

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.

141345

Recommendation: keep the original judging.

> Escalate for 10 USDC The pools used in the margin dex are whitelisted and therefore highly liquid. Using a flashloan in such a pool to manipulate the reserves would not be economically viable since the flashloans are attached with a fee. It would not even be possible to repay the loan in the scenario described in the report (it would revert).
>
> This issue should be at most medium.
>
> PS: Note that it is not possible to sandwich other's transactions on Arbitrum. So the above assumes that a malicious user wants to sandwich its own transaction by making several function call in one transaction. I'm no MEV expert, so correct me if I'm wrong, but Arbitrum uses a sequencer to order transactions and does not have a mempool. Instead, transactions are ordered on a first come, first served basis. Therefore, it is only possible to back-run transactions. On Arbitrum MEV bots are competing for the lowest latency to the sequencer in order to identify profitable transactions and initiate a back-run.
>
> The pools used in the margin dex are whitelisted

Whitelisted pool can not guarantee liquidity all the time. With flashloan, the liquidity could be manipulated. The profit is the loss from the LP here, which can cover the flashloan fee.

This vector is suitable for self sandwich, the attack is by design, abusing the delayed `isLiquidatable()` check.

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/13

hrishibhat

Result: High Has duplicates After further discussions with the Lead judge, Agree that this could be on the border with respect to severity. Based on the points raised in issues #140, and #106, and duplicates such as #54. With all the information presented , considering this issue a valid high.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- twicek: rejected

maarcweiss

Fixed by calling `self._is_liquidatable(position_uid)` after opening and reducing a position.

Issue H-4: `reduce_position` doesn't update margin mapping correctly

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/143

Found by

0xbepresent, Dug, GimelSec, Yuki, n33k, twicek

Summary

`reduce_position` function decrease the margin amount of the position but doesn't add it back to the user's margin mapping, making it impossible to withdraw the margin.

Vulnerability Detail

After selling some position tokens back against debt tokens using `reduce_position` function, `debt_shares` and `margin_amount` are reduced proportionally to keep leverage the same as before:

Vault.vy#L313-L330

```
debt_amount: uint256 = self._debt(_position_uid)
    margin_debt_ratio: uint256 = position.margin_amount * PRECISION / debt_amount


    amount_out_received: uint256 = self._swap(
        position.position_token, position.debt_token, _reduce_by_amount,
↪   min_amount_out
    )


    # reduce margin and debt, keep leverage as before
    reduce_margin_by_amount: uint256 = (
        amount_out_received * margin_debt_ratio / PRECISION
    )
    reduce_debt_by_amount: uint256 = amount_out_received -
↪   reduce_margin_by_amount


    position.margin_amount -= reduce_margin_by_amount
```

```
    burnt_debt_shares: uint256 = self._repay(position.debt_token,
↪   reduce_debt_by_amount)
    position.debt_shares -= burnt_debt_shares
    position.position_amount -= _reduce_by_amount
```

However, even though some of the margin have been paid back (`position.margin_amount` has been reduced), `self.margin[position.account][position.debt_token]` mapping hasn't been updated by adding `reduce_margin_by_amount` which would allow the user to withdraw his margin.

Impact

Users will lose their margin tokens.

Code Snippet

Vault.vy#L313-L330

Tool used

Manual Review

Recommendation

Consider modifying the code like this:

```
    reduce_debt_by_amount: uint256 = amount_out_received -
↪   reduce_margin_by_amount


    position.margin_amount -= reduce_margin_by_amount
+   self.margin[position.account][position.debt_token] += reduce_margin_by_amount

    burnt_debt_shares: uint256 = self._repay(position.debt_token,
↪   reduce_debt_by_amount)
    position.debt_shares -= burnt_debt_shares
    position.position_amount -= _reduce_by_amount
```

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/6

maarcweiss

Fixed by adding back margin to the user's margin mapping while reducing the position

Issue H-5: Leverage calculation is wrong

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/150

Found by

0xDjango, n33k, twicek

Summary

Leverage calculation is wrong which will lead to unfair liquidations or over leveraged positions depending on price movements.

Vulnerability Detail

`_calculate_leverage` miscalculate the leverage by using `_debt_value + _margin_value` as numerator instead of `_position_value` :

Vault.vy#L465-L477

```
def _calculate_leverage(
    _position_value: uint256, _debt_value: uint256, _margin_value: uint256
) -> uint256:
    if _position_value <= _debt_value:
        # bad debt
        return max_value(uint256)


    return (
        PRECISION
        * (_debt_value + _margin_value)
        / (_position_value - _debt_value)
        / PRECISION
    )
```

The three inputs of the function `_position_value`, `_debt_value` and `_margin_value` are all determined by a chainlink oracle price feed. `_debt_value` represents the value of the position's debt share converted to debt amount in USD. `_margin_value` represents the current value of the position's initial margin amount in USD. `_position_value` represents the current value of the position's initial position amount in USD.

The problem with the above calculation is that `_debt_value + _margin_value` does not represent the value of the position. The leverage is the ratio between the current value of the position and the current margin value. `_position_value - _debt_value` is correct and is the current margin value, but `_debt_value +`

`_margin_value` doesn't represent the current value of the position since there is no guarantee that the debt token and the position token have correlated price movements.

Example: debt token: ETH, position token: BTC.

- Alice uses 1 ETH of margin to borrow 14 ETH (2k USD/ETH) and get 1 BTC (30k USD/BTC) of position token. Leverage is 14.
- The next day, the price of ETH in USD is still 2k USD/ETH but BTC price in USD went down from 30k to 29k USD/BTC. Leverage is now (`_position_value == 29k`) / (`_position_value == 29k - _debt_value == 28k`) = 29, instead of what is calculated in the contract: (`_debt_value == 28k + _margin_value == 2k`) / (`_position_value == 29k - _debt_value == 28k`) = 30.

Impact

Leverage calculation is wrong which will lead to unfair liquidations or over leveraged positions depending on price movements.

Code Snippet

Vault.vy#L465-L477

Tool used

Manual Review

Recommendation

Consider modifying the code like this:

```
def _calculate_leverage(
    _position_value: uint256, _debt_value: uint256, _margin_value: uint256
) -> uint256:
    if _position_value <= _debt_value:
        # bad debt
        return max_value(uint256)


    return (
        PRECISION
-       * (_debt_value + _margin_value)
+       * (_position_value)
        / (_position_value - _debt_value)
        / PRECISION
    )
```

Discussion

Unstoppable-DeFi

While our implementation uses a slightly different definition of leverage and the initial margin + debt as base, we agree that the above implementation is cleaner and more intuitive for users. Both formulas would work similarly though and liquidate positions timely to protect the protocol.

141345

agree, maybe medium is more appropriate

141345

The dup https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/100 gives an example why the current formula could lead to unexpected result.

twicek

Escalate for 10 USDC. My report shows why the current used formula is wrong as it does not take into account that debt tokens and position tokens are not necessarily tokens with correlated prices. The duplicate #100 shows in another way that the formula fail to calculate the leverage of a position correctly. The impact is the same, but my report highlight `_debt_value + _margin_value != _position_value`, the same way that the debt against a house is not equal to the market value of this house (also described in another way in #156). The definition of leverage used in the code is not correct and will lead to unfair liquidations or over leveraged positions, which is definitely high severity.

sherlock-admin2

> Escalate for 10 USDC. My report shows why the current used formula is wrong as it does not take into account that debt tokens and position tokens are not necessarily tokens with correlated prices. The duplicate #100 shows in another way that the formula fail to calculate the leverage of a position correctly. The impact is the same, but my report highlight `_debt_value + _margin_value != _position_value`, the same way that the debt against a house is not equal to the market value of this house (also described in another way in #156). The definition of leverage used in the code is not correct and will lead to unfair liquidations or over leveraged positions, which is definitely high severity.

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.

fatherGoose1

The duplicate report #100 shows how adding margin makes a position less healthy, and removing margin makes a position healthier. This is simply a backwards implementation that will lead to unfair liquidations or higher leverage than should be possible. The impact is clear loss of funds.

141345

Recommendation: change the original judging, to high severity.

> Escalate for 10 USDC. My report shows why the current used formula is wrong as it does not take into account that debt tokens and position tokens are not necessarily tokens with correlated prices. The duplicate #100 shows in another way that the formula fail to calculate the leverage of a position correctly. The impact is the same, but my report highlight `_debt_value + _margin_value != _position_value`, the same way that the debt against a house is not equal to the market value of this house (also described in another way in #156). The definition of leverage used in the code is not correct and will lead to unfair liquidations or over leveraged positions, which is definitely high severity.

Unexpected and unfair liquidation could cause loss to users. Since the issue roots from the formula, the loss could be long term, result in accumulated fund loss for users, can can be deemed as "material loss of funds".

Based on the above, high severity might be appropriate.

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/14

hrishibhat

@Unstoppable-DeFi based on the above escalation it seems to be a high issue. Is there any other reason this should not be a high-severity issue?

hrishibhat

Result: High Has duplicates Considering this issue a valid high

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- twicek: accepted

maarcweiss

Fixed by changing the leverage calculation formula from using `* (_debt_value + _margin_value)` to using `* (_position_value)` instead

Issue H-6: Vault: `_update_debt` does not accrue interest

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/167

Found by

0x00ffDa, 0xDjango, 0xStalin, 0xpinky, Bauer, BugBusters, Dug, GimelSec, kutugu, n33k, pengun

Summary

`_update_debt` call `_debt_interest_since_last_update` to accrue interest but `_debt_interest_since_last_update` always return 0 in `_update_debt`.

Vulnerability Detail

`_update_debt` sets `self.last_debt_update[_debt_token]` to `block.timestamp` and then calls `_debt_interest_since_last_update`.

```
def _update_debt(_debt_token: address):
    ....
    self.last_debt_update[_debt_token] = block.timestamp
    ....
    self.total_debt_amount[_debt_token] += self._debt_interest_since_last_update(
        _debt_token
    )
```

`_debt_interest_since_last_update` always returns 0. because `block.timestamp - self.last_debt_update[_debt_token]` is always 0.

```
def _debt_interest_since_last_update(_debt_token: address) -> uint256:
    return (
        (block.timestamp - self.last_debt_update[_debt_token])
        * self._current_interest_per_second(_debt_token)
        * self.total_debt_amount[_debt_token]
        / PERCENTAGE_BASE
        / PRECISION
    )
```

Impact

Debt fee is not accrued.

Code Snippet

https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L1050-L1076

Tool used

Manual Review

Recommendation

Call `_debt_interest_since_last_update` then update `last_debt_update`.

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/7

0x00ffDa

Escalate for 10 USDC.

I believe this should be high severity. This flaw results in definite loss of funds for the protocol and LP providers: zero revenue from all debts, no special circumstances needed.

sherlock-admin2

> Escalate for 10 USDC.
>
> I believe this should be high severity. This flaw results in definite loss of funds for the protocol and LP providers: zero revenue from all debts, no special circumstances needed.

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.

141345

Recommendation: keep the original judging.

> Escalate for 10 USDC.
>
> I believe this should be high severity. This flaw results in definite loss of funds for the protocol and LP providers: zero revenue from all debts, no special circumstances needed.

The loss is on interest, not big loss in principal, the amount might not be considered "material loss of funds".

I think medium is appropriate.

Nabeel-javaid

IMO, high is appropriate is it is bricking the core functionality of interest. There is no point of lending when there is no interest on it.

0xpinky

interest is core concept in lend and borrow, flawed interest updates would brick one of the core functions. when we look at this over a period of time, the loss would have incurred is huge.. High is appropriate.

hrishibhat

Result: High Has duplicates Considering this issue a valid high for two reasons:

- It breaks the core functionality of accruing interest on debt.
- Leading loss of revenue for protocol and LP.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- 0x00ffDa: accepted

maarcweiss

Fixed by calling `last_debt_update` after `_debt_interest_since_last_update`

Issue H-7: Adversary manipulate the middle path when calling `execute_dca_order`, resulting user loss, benefiting the attacker

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/182

Found by

0xDjango, 0xyPhilic, BugBusters, Lalanda, chainNue, pengun

Summary

Adversary manipulate the middle path when calling `execute_dca_order`, resulting user loss, benefiting the attacker

Vulnerability Detail

Allowing anyone to call the `execute_dca_order` function with a custom `_uni_hop_path` introduce security issue. If an attacker constructs a malicious path with their own token in the middle, they could manipulate the liquidity and perform an exploit. The `_uni_hop_path` parameter in `execute_dca_order` is really dangerous input which can be manipulated by attacker.

For example a user want to Dca `USDC -> WETH` by posting `post_dca_order` with `_token_in` is USDC, and `_token_out` is WETH, and with the `min_amount_out` generated from `_calc_min_amount_out` which is based on twap and slippage percentage (in contrary, `LimitOrders.vy` use input `min_amount_out` for token out from the `post_limit_order` function, thus will not facing same effect like this issue)

When attacker see this Dca order, they can create a new ERC20 `MYTOKEN` (and will provide liquidity in uniswap) and plan to attack the Dca by selecting his own token as the middle path, so `execute_dca_order` will be executed using `USDC -> MYTOKEN -> WETH` path.

Since attacker can manipulate the liquidity of `MYTOKEN` in Uniswap, resulting user will get bad swap amount due to the custom path provided by attacker.

Slippage and twap length as protection are not enough, because the middle token path is the problem here. Moreover, using twap with twap_length can also be manipulated by preparing / pre-attack pool since the `last_execution` and `seconds_between_executions` is known, also the order's twap_length is visible, so attack scenario can prepared beforehand.

```
File: Dca.vy
163: @external
```

```
164: @nonreentrant('lock')
165: def execute_dca_order(_uid: bytes32, _uni_hop_path: DynArray[address, 3],
↪  _uni_pool_fees: DynArray[uint24, 2], _share_profit: bool):
...
174:     # ensure path is valid
175:     assert len(_uni_hop_path) in [2, 3], "[path] invlid path"
176:     assert len(_uni_pool_fees) == len(_uni_hop_path)-1, "[path] invalid
↪  fees"
177:     assert _uni_hop_path[0] == order.token_in, "[path] invalid token_in"
178:     assert _uni_hop_path[len(_uni_hop_path)-1] == order.token_out, "[path]
↪  invalid token_out"
...
206:     # Vyper way to accommodate abi.encode_packed
207:     path: Bytes[66] = empty(Bytes[66])
208:     if(len(_uni_hop_path) == 2):
209:         path = concat(convert(_uni_hop_path[0], bytes20),
↪  convert(_uni_pool_fees[0], bytes3), convert(_uni_hop_path[1], bytes20))
210:     elif(len(_uni_hop_path) == 3):
211:         path = concat(convert(_uni_hop_path[0], bytes20),
↪  convert(_uni_pool_fees[0], bytes3), convert(_uni_hop_path[1], bytes20),
↪  convert(_uni_pool_fees[1], bytes3), convert(_uni_hop_path[2], bytes20))
212:
213:     min_amount_out: uint256 =
↪  self._calc_min_amount_out(order.amount_in_per_execution, _uni_hop_path,
↪  _uni_pool_fees, order.twap_length, order.max_slippage)
214:
215:     uni_params: ExactInputParams = ExactInputParams({
216:         path: path,
217:         recipient: self,
218:         deadline: block.timestamp,
219:         amountIn: order.amount_in_per_execution,
220:         amountOutMinimum: min_amount_out
221:     })
222:     amount_out: uint256 =
↪  UniswapV3SwapRouter(UNISWAP_ROUTER).exactInput(uni_params)
...
238:
```

Steps:

1. A malicious user creates a custom token called "MYTOKEN" and provides liquidity for it on Uniswap v3. They allocate liquidity to create extreme price ranges for MYTOKEN.
2. A regular user intends to swap USDC to WETH.
3. The malicious user intercepts the transaction and modifies the path by adding intermediate token with MYTOKEN. The new path becomes USDC ->

MYTOKEN -> WETH.

4. Uniswap v3 receives the modified path and attempts to execute the swap based on the provided path.
5. Uniswap v3 calculates the best available price based on the modified path, which includes MYTOKEN.
6. Due to the manipulated liquidity pool of MYTOKEN, the price of MYTOKEN is significantly skewed, leading to an unfair price for the regular user executing the multi-hop swap.
7. The malicious user takes advantage of the distorted liquidity pool by executing multi-hop swaps involving MYTOKEN at highly advantageous prices.
8. Innocent users who unknowingly interact with the manipulated liquidity pool may receive unfavorable prices for their USDC to WETH swaps, leading to financial losses.

In this scenario, the malicious user modifies the path of the multi-hop swap by replacing the intended intermediate token with MYTOKEN. As a result, the malicious user exploits the manipulated liquidity pool and executes trades at favorable prices, while causing losses for other traders and disrupting the market equilibrium

Impact

User will get a bad rate (far from the normal rate) due to middle token rate is manipulated, thus losing his asset.

Code Snippet

https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/spot-dex/Dca.vy#L163-L237

Tool used

Manual Review

Recommendation

Might need to have a minimal desired output amount of range (for the output token), or remove the option to input manual path with anyone can call the `execute_dca_order` then replace it with the Oracle price rate. Other way, consider implement a whitelist token for swap path.

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/18

maarcweiss

Fixed by adding a whitelist of tokens that users can swap from. Although fixed, I see that tokens have been hardcoded in the contract. It would be better to not hardcode them and add a setter function for the whitelisting of those tokens.

Issue H-8: Interested calculated is ampliefied by multiple of 1000 in `_debt_interest_since_last_update`

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/191

Found by

BugBusters, twicek

Summary

Interest calculated in the `_debt_interest_since_last_update` function is amplified by multiple of 1000, hence can completely brick the system and debt calculation. Because we divide by PERCENTAGE_BASE instead of PERCENTAGE_BASE_HIGH which has more precision and which is used in utilization calculation.

Vulnerability Detail

Following function calculated the interest accured over a certain interval :

```
def _debt_interest_since_last_update(_debt_token: address) -> uint256:

    return (

        (block.timestamp - self.last_debt_update[_debt_token])*
↪   self._current_interest_per_second(_debt_token)
        * self.total_debt_amount[_debt_token]
        / PERCENTAGE_BASE
        / PRECISION
    )
```

But the results from the above function are amplified by factor of 1000 due to the reason that the intrest per second as per test file is calculated as following:

```
# accordingly the current interest per year should be 3% so 3_00_000
# per second that is (300000*10^18)/(365*24*60*60)
expected_interest_per_second = 9512937595129375

assert (
    expected_interest_per_second
    == vault_configured.internal._current_interest_per_second(usdc.address)
)
```

So yearly interest has the precision of 5 as it is calculated using utilization rate and `PERCENTAGE_BASE_HIGH_PRECISION` is used which has precision of 5 .and per second has the precision of 18, so final value has the precision of 23.

Interest per second has precision = 23.

But if we look at the code:

```
(block.timestamp - self.last_debt_update[_debt_token])*
↪  self._current_interest_per_second(_debt_token)
* self.total_debt_amount[_debt_token]
/ PERCENTAGE_BASE
/ PRECISION
```

We divide by PERCENTAGE_BASE that is = 100_00 = precision of => 2 And than by PRECISION = 1e18 => precision of 18. So accumulated precision of 20, where as we should have divided by value precises to 23 to match the nominator.

Where is we should have divided by PERCENTAGE_BASE_HIGH instead of PERCENTAGE_BASE

Hence the results are amplified by enormous multiple of thousand.

Impact

Interest are too much amplified, that impacts the total debt calculation and brick whole leverage, liquidation and share mechanism.

Note: Dev confirmed that the values being used in the tests are the values that will be used in production.

Code Snippet

https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L1069-L1076 https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/tests/vault/test_variable_interest_rate.py#L314-L333

Tool used

Manual Review

Recommendation

Use PERCENTAGE_BASE_HIGH in division instead of PERCENTAGE_BASE.

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/8

Nabeel-javaid

Escalate for 10 USDC

This should be high as described impact in the given submission and the duplicate too.

sherlock-admin2

> Escalate for 10 USDC
>
> This should be high as described impact in the given submission and the duplicate too.

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.

twicek

Escalate for 10 USDC The wrong calculation of interest rates will cause a direct loss of funds to users. This should definitely be high severity.

sherlock-admin2

> Escalate for 10 USDC The wrong calculation of interest rates will cause a direct loss of funds to users. This should definitely be high severity.

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.

141345

Recommendation: change the original judging, to high severity.

> Escalate for 10 USDC
>
> This should be high as described impact in the given submission and the duplicate too.

A magnitude of 1000 times of interest can be deemed as "material loss of funds".

141345

> Escalate for 10 USDC The wrong calculation of interest rates will cause a direct loss of funds to users. This should definitely be high severity.

Same as above

hrishibhat

Result: High Has duplicates Considering this a valid high

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- Nabeel-javaid: accepted
- twicek: accepted

maarcweiss

Fixed by calling PERCENTAGE_BASE_HIGH instead of PERCENTAGE_BASE for the interest calculation

Issue M-1: Debt is not updated when removing margin from a position

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/9

Found by

stopthecap

Summary

Debt is not updated when removing margin from a position

Vulnerability Detail

Traders are allowed to remove margin/collateral from their positions: https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L528-L546

as you can see, the position can only have collateral removed if it is not liquidable: https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L487

The problem is that the total `total_debt_shares[_debt_token]` is not updated when checking whether the positon is liquidable neither when calling `remove_margin` You can see on the following links the progression of the function calls to calculate whether a position is liquidable or not:

https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L444

https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L1142

https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L1099

https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L1111

https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L1117C16-L1117C16

As seen, `total_debt_shares[_debt_token]` is not updated to the current `total_debt_shares[_debt_token]` by calling `_update_debt` and therefore a stale `total_debt_shares[_debt_token]` is used to calculate whether a position is liquidable. Allowing positions that are in fact liquidable remove margin because the debt is not updated.

Impact

Not updating the debt before removing collateral (margin) from your position does allow a trader to remove collateral from a liquidable position

Code Snippet

https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L528-L546

https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L1117C16-L1117C16

Tool used

Manual Review

Recommendation

Call `_update_debt` at the beginning of the remove margin function

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/1

maarcweiss

Fixed by calling update debt when removing margin from positions

Issue M-2: _account_for_withdraw_liquidity rounding in the wrong direction will run out of the vault

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/37

Found by

kutugu

Summary

Vault implementers should be aware of the need for specific, opposing rounding directions across the different mutable and view methods, as it is considered most secure to favor the Vault itself during calculations over its users If (1) it's calculating how many shares to issue to a user for a certain amount of the underlying tokens they provide or (2) it's determining the amount of the underlying tokens to transfer to them for returning a certain amount of shares, it should round down. If (1) it's calculating the amount of shares a user has to supply to receive a given amount of the underlying tokens or (2) it's calculating the amount of underlying tokens a user has to provide to receive a certain amount of shares, it should round up.

When the user extracts liquidity, due to the existence of share, the calculation has precision errors, should pay special attention to the rounding direction. Users withdraw a specific amount of liquidity, user shares should be rounded up, subtract 1 more shares, to avoid spending more than they earn, resulting in the eventual run out of the vault.

Vulnerability Detail

`_amount_to_lp_shares` calculates the amount of share to be subtracted, which should round up, otherwise the user account share will be 1 more than it actually is, and the difference will gradually become larger as users interact with vault over time, eventually run out of the vault.

Impact

Rounding in the wrong direction, when users withdraw liquidity, the shares are greater than actual value, and gradually accumulate as interactions, eventually causing the vault to be run out of and unable to repay all the liquidity.

Code Snippet

- https://github.com/sherlock-audit/2023-06-unstoppable/blob/94a68e49971bc6942c75da76720f7170d46c0150/unstoppable-dex-audit/contracts/margin-

dex/Vault.vy#L893-L912

Tool used

Manual Review

Recommendation

Round up in _amount_to_lp_shares

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/17

maarcweiss

Fixed by subtracting a share from the calculation of `self._amount_to_lp_shares` to account for the exposed rounding error

Issue M-3: Spot dex cant handle fee-on-transfer tokens

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/40

Found by

0x4non, 0xStalin, kutugu, twicek

Summary

`Dca.vy`, `LimitOrder.vy` and `TrailingStopDex.vy` does not properly support tokens that implement a fee on transfer. These types of tokens deduct a fee from any transfer of tokens, resulting in the recipient receiving less than the original amount sent.

Vulnerability Detail

The specific issue lies in the ERC20 token transfer.

Lests analyze the method `execute_limit_order` focus on lines LimitOrders.vy#L160-L161, this code assumes that the full `order.amount_in` will be transferred to `self`. This assumption is incorrect when the token in question implements a fee on transfer. When a fee is deducted, `self` will receive less tokens than `order.amount_in`. The subsequent call to `approve` and `ExactInputSingleParams` could therefore potentially fail as they rely on `self` having a balance of `order.amount_in`.

Impact

This vulnerability could potentially halt token swaps midway if the token involved deducts a transfer fee. This can result in an unsuccessful token swap, which in turn could lock funds and possibly lead to financial loss for the user.

Code Snippet

- Dca.vy#L200-L201
- LimitOrders.vy#L160-L161
- TrailingStopDex.vy#L170-L171

Tool used

Manual Review

Recommendation

To rectify this vulnerability, it's recommended to replace `order.amount_in` with the account of balance of `ERC20(order.token_in)` after the swap minus before the transferFrom call. This ensures that the correct balance (after any potential fees) is used for the approve call and the ExactInputSingleParams. It's also advised to add error handling for unsuccessful transfers and approvals.

Example:

Discussion

dot-pengun

Escalate

The contest page states that only tokens supported by uniswap v3 can be used, as shown below.

> The Spot contracts need to be able to interact with any pool/token, we don’t want to have a centralized whitelist, we want it to work with any Uniswap v3 pool (and later others but this is probably out of scope).

According to the uniswap V3 docs, uniswap do not support fee-on-transfer tokens, so I believe this issue is invalid.

sherlock-admin2

> Escalate
>
> The contest page states that only tokens supported by uniswap v3 can be used, as shown below.
>
> > The Spot contracts need to be able to interact with any pool/token, we don’t want to have a centralized whitelist, we want it to work with any Uniswap v3 pool (and later others but this is probably out of scope).
>
> According to the uniswap V3 docs, uniswap do not support fee-on-transfer tokens, so I believe this issue is invalid.

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.

twicek

Escalate for 10 USDC This sentence implies that any token could be interacted with in the spot contracts:

> The Spot contracts need to be able to interact with any pool/token, we don’t want to have a centralized whitelist, we want it to work with any Uniswap v3 pool (and later others but this is probably out of scope).

The sponsor didn't know that fee-on-transfer tokens couldn't be used in UniswapV3 when writing the README in which fee-on-transfer token related issues are explicitly in scope for the spot dex:

> Q: Are there any FEE-ON-TRANSFER tokens interacting with the smart contracts? Margin: no Spot: possibly

Information acquired after the beginning of the contest and not explicitly mentioned in the README shouldn't supersede the README information.

sherlock-admin2

> Escalate for 10 USDC This sentence implies that any token could be interacted with in the spot contracts:
>
> > The Spot contracts need to be able to interact with any pool/token, we don’t want to have a centralized whitelist, we want it to work with any Uniswap v3 pool (and later others but this is probably out of scope).
>
> The sponsor didn't know that fee-on-transfer tokens couldn't be used in UniswapV3 when writing the README in which fee-on-transfer token related issues are explicitly in scope for the spot dex:
>
> > Q: Are there any FEE-ON-TRANSFER tokens interacting with the smart contracts? Margin: no Spot: possibly
>
> Information acquired after the beginning of the contest and not explicitly mentioned in the README shouldn't supersede the README information.

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.

141345

Recommendation: keep the original judging.

> Escalate
>
> The contest page states that only tokens supported by uniswap v3 can be used, as shown below.
>
> > The Spot contracts need to be able to interact with any pool/token, we don’t want to have a centralized whitelist, we

> want it to work with any Uniswap v3 pool (and later others but this is probably out of scope).
>
> According to the uniswap V3 docs, uniswap do not support fee-on-transfer tokens, so I believe this issue is invalid.

The README is a little ambiguous.

The designed behavior seems to be compatible with fee on transfer token. And the POC demonstrates that the current codebase needs improvement for this purpose.

141345

> The README is a little ambiguous.
>
> The designed behavior seems to be compatible with fee on transfer token. And the POC demonstrates that the current codebase needs improvement for this purpose.

same as above

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/10

hrishibhat

Result: Medium Has duplicates Agree with the Lead judge's comment below:

> The README is a little ambiguous. The designed behavior seems to be compatible with fee on transfer token. And the POC demonstrates that the current codebase needs improvement for this purpose.

Accepting the escalation because it is understandable that it could be confusing, but will keep the issue valid medium.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- dot-pengun: accepted
- twicek: accepted

maarcweiss

Fixed by using the common pattern of checking balances before and after the transfers and calculating the real amount in spot dex

Issue M-4: `Vault._amount_per_base_lp_share` should also consider bad debt when `safety_module_lp_total_amount` is not enough

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/92

Found by

GimelSec

Summary

Unstoppable's vault consists of two types of pools. The safety module pool is designed to bear the first loss risk. So the bad debt is first covered by the safety module liquidity. https://github.com/sherlock-audit/2023-06-unstoppable-sces60107/tree/main/unstoppable-dex-audit#margin

> Liquidity Providers have the choice to provide liquidity in the "Base LP" pool or the "Safety Module" pool. The Safety Module pool takes a first loss risk protecting the Base LP in exchange for a higher share of the trading and accruing interest.

However, if the safety module liquidity is insufficient, the base liquidity is also used to cover the bad debt. But `Vault._amount_per_base_lp_share` never take bad debt into consideration, leading to loss of funds.

Vulnerability Detail

When calling `Vault.withdraw_liquidity`, it uses `Vault._account_for_withdraw_liquidity` to calculate the withdraw share. https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L884

If the base pool is chosen, it calls `self._amount_per_base_lp_share`. And it directly uses `self.base_lp_total_amount` to calculate the shares without considering the bad debt. https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L910 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L946

Suppose that `base_lp_total_amount` is 100 and `safety_module_lp_total_amount` is 100. The `bad_debt` is now 150 and `total_debt_amount` is 0. The available liquidity is 50. https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L989

Consider the following situations.

- The total share of base liquidity is 100 shares, with Alice and Bob each having 50 shares.
- Alice calls `withdraw_liquidity` to withdraw 50 tokens from the base pool.
- Bob also wants to withdraw the liquidity but he fails since the available liquidity is 0 now. https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L863

Impact

If the amount of bad debt is more than safe module liquidity, the withdrawal of base liquidity becomes unfair. Early liquidity providers have the advantage of being able to withdraw the full amount, while other liquidity providers are unable to withdraw any liquidity, resulting in a loss of funds for them.

Code Snippet

https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L946 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L957 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L967 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L884 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L910 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L863 https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L989

Tool used

Manual Review

Recommendation

If the bad debt exceeds the available safety module liquidity, `_amount_per_base_lp_share` should take bad debt into consideration.

```
def _amount_per_base_lp_share(_token: address) -> uint256:
    return (
-       self.base_lp_total_amount[_token]
+       self._base_total_amount(_token)
        * PRECISION
        * PRECISION
```

```
        / self.base_lp_total_shares[_token]
    )

+@internal
+@view
+def _base_total_amount(_token: address) -> uint256:
+   if self.bad_debt[_token] > self.safety_module_lp_total_amount[_token] +
 ↪  self.base_lp_total_amount[_token]:
+       return 0
+   if self.bad_debt[_token] <= self.safety_module_lp_total_amount[_token]:
+       return self.base_lp_total_amount[_token]
+   return self.safety_module_lp_total_amount[_token] +
 ↪  self.base_lp_total_amount[_token] - self.bad_debt[_token]
```

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/16

maarcweiss

Introduced a new view function to account for when the bad debt surpasses the amount covered by `safety_module`. Missing

```
+ if self.bad_debt[_token] > self.safety_module_lp_total_amount[_token] +
 ↪  self.base_lp_total_amount[_token]:
+       return 0
```

from watsons recommendation though. Seems good to me but I would like @Unstoppable-DeFi to re-check if letting this check out was intended

Unstoppable-DeFi

Yes, intended.

The definition of bad debt is the amount of liquidity that was borrowed to open a position and not recovered when closing the position (offset by the traders provided margin). Therefore the maximum amount of bad debt that could theoretically be accrued is the total amount of liquidity available (`self.safety_module_lp_total_amount[_token] + self.base_lp_total_amount[_token]`). There is no scenario where bad debt could be greater than this since closing a position cannot end in a negative return (i.e. is clamped to `0`).

maarcweiss

> Introduced a new view function to account for when the bad debt surpasses the amount covered by `safety_module`. Missing

```
+ if self.bad_debt[_token] > self.safety_module_lp_total_amount[_token] +
↪  self.base_lp_total_amount[_token]:
+       return 0
```

from watsons recommendation though. Seems good to me but I would like @Unstoppable-DeFi to re-check if letting this check out was intended

Following the sign-off, the previously explained snippet was not included because bad debt can't actually surpass: `self.safety_module_lp_total_amount[_token] + self.base_lp_total_amount[_token]` therefore the case is not accounted for in the contracts

Issue M-5: The `Vault._update_debt()` function should be executed before admin sets new interest rate via `Vault.set_variable_interest_parameters()`

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/103

Found by

0xbepresent

Summary

The Vault._update_debt() should be executed before the interests rate are modified by the Vault.set_variable_interest_parameters() function.

Vulnerability Detail

The Vault._update_debt() function helps to accrue interest since the last update. The execution path:

1. It calls the function Vault._debt_interest_since_last_update(_debt_token) in order to calculate the token debt interest.
2. The `Vault._debt_interest_since_last_update()` function uses the Vault._current_interest_per_second() function in order to get the interest per second.
3. The Vault._current_interest_per_second() calls Vault._interest_rate_by_utilization() in order to get the `interest rate`.
4. The Vault._interest_rate_by_utilization() function calls the Vault._dynamic_interest_rate_low_utilization() or Vault._dynamic_interest_rate_high_utilization depending on the `switch utilization`.
5. The Vault._dynamic_interest_rate_low_utilization() is executed and it calls the Vault._min_interest_rate().
6. Then, the Vault._min_interest_rate() function gets the value from the [self.interest_configuration_address][0]

So in the step 6, it gets the interest from the `interest_configuration`, then the interests are used in order to calculate the token debt. The `Vault._update_debt()` accrue interest since the last update, the function should be executed before the admin change the interest configuration via Vault.set_variable_interest_parameters() function. The reason is that the time that has already passed must be taken with the previous interest before the admin changes to the new interests rate.

Impact

The protocol should accrue interest since the last update before change the interest rate configuration. If the admin change the interest rate, the new interest rate should be applied to the future time not to the time that has already passed. The time that has already passed should use the previous interest.

It is unfair for the users because users expects interests changes to be applied after they were changed not to the past time the `_update_debt` has not been executed.

Code Snippet

- The Vault._update_debt() function.
- The Vault._debt_interest_since_last_update function.
- The Vault._current_interest_per_second() function.
- The Vault._interest_rate_by_utilization() function.
- The Vault._dynamic_interest_rate_low_utilization() function.
- The Vault._dynamic_interest_rate_high_utilization() function.
- The Vault._interest_configuration() function.
- The Vault.set_variable_interest_parameters() function.

Tool used

Manual review

Recommendation

Execute `self._update_debt(address)` before the interests are modified:

```
@external
def set_variable_interest_parameters(
    _address: address,
    _min_interest_rate: uint256,
    _mid_interest_rate: uint256,
    _max_interest_rate: uint256,
    _rate_switch_utilization: uint256,
):
    assert msg.sender == self.admin, "unauthorized"
++  self._update_debt(address)
    self.interest_configuration[_address] = [
        _min_interest_rate,
        _mid_interest_rate,
        _max_interest_rate,
```

```
        _rate_switch_utilization,
    ]
```

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/4

maarcweiss

Fixed by calling update_debt before setting new interest params in the `set_variable_interest_parameters` function

Issue M-6: The `Vault._to_usd_oracle_price()` function uses the same `ORACLE_FRESHNESS_THRESHOLD` for all token prices feeds which is incorrect

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/104

Found by

0xbepresent, 0xpinky, TheNaubit, n33k, pengun, stopthecap, twicek

Summary

The same `ORACLE_FRESHNESS_THRESHOLD` is used for all the token prices feeds which can be dangerous because different pairs of tokens have different freshness intervals.

Vulnerability Detail

The ORACLE_FRESHNESS_THRESHOLD is 24 hours constant. It is used to check if the Oracle price is fresh in the 580 code line.

```
File: Vault.vy
562: def _to_usd_oracle_price(_token: address) -> uint256:
...
...
576:     round_id, answer, started_at, updated_at, answered_in_round =
↪  ChainlinkOracle(
577:         self.to_usd_oracle[_token]
578:     ).latestRoundData()
579:
580:     assert (block.timestamp - updated_at) < ORACLE_FRESHNESS_THRESHOLD,
↪  "oracle not fresh"
...
...
```

The problem is that different pairs have diferrent `heartbeats`. For example the LINK/USD has a `heartbeat` of 3600 seconds so since the `ORACLE_FRESHNESS_THRESHOLD` is set to `24 hours`, the check for `LINK/USD` is useless since the its hearbeat is 3600 seconds. The same behaivour in CRV/USD which has a heartbeat of 3600 seconds.

Impact

Using the same `ORACLE_FRESHNESS_THRESHOLD` (heartbeat) for all the price feeds is not correct becuase the freshness validation would be useless for some pairs

which can return stale data.

Code Snippet

The Vault._to_usd_oracle_price() function:

```
File: Vault.vy
562: def _to_usd_oracle_price(_token: address) -> uint256:
563:     """
564:     @notice
565:         Retrieves the latest Chainlink oracle price for _token.
566:         Ensures that the Arbitrum sequencer is up and running and
567:         that the Chainlink feed is fresh.
568:     """
569:     assert self._sequencer_up(), "sequencer down"
570:
571:     round_id: uint80 = 0
572:     answer: int256 = 0
573:     started_at: uint256 = 0
574:     updated_at: uint256 = 0
575:     answered_in_round: uint80 = 0
576:     round_id, answer, started_at, updated_at, answered_in_round =
 ↪  ChainlinkOracle(
577:         self.to_usd_oracle[_token]
578:     ).latestRoundData()
579:
580:     assert (block.timestamp - updated_at) < ORACLE_FRESHNESS_THRESHOLD,
 ↪  "oracle not fresh"
581:
582:     usd_price: uint256 = convert(answer, uint256)  # 8 dec
583:     return usd_price
```

Tool used

Manual review

Recommendation

Use the corresponding heartbeat `ORACLE_FRESHNESS_THRESHOLD` for each token in the `Vault._to_usd_oracle_price()` function.

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/11

maarcweiss

Fixed by introducing a mapping that has an individual `ORACLE_FRESHNESS_THRESHOLD` for each token, instead of using the same one for all of them.

Issue M-7: Vault multihop swaps for any token pair will fail in one direction

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/108

Found by

0x00ffDa

Summary

The Unstoppable SwapRouter's add_path() function stores the same swap path for swapping from "token1" to "token2" and from "token2" to "token1". If the swap path starts with the token1 address, swaps from token2 to token1 will fail, and vice versa.

Vulnerability Detail

The administrator must call `SwapRouter` `add_path()` to add support for swaps that use multiple Uniswap pools (multihop swaps). The path provided is a sequence of token addresses and pool fee amounts. The path is stored in the `self.paths` hashmap using the provided `token1` and `token2` addresses as the keys. The path is stored twice, once for each ordering of those keys such that the stored path can be accessed easily without sorting the keys.

When a path stored in the Unstoppable Vault is used for a swap, it is obtained by indexing the hashmap first with the address of the input token for the swap:

```
path: Bytes[66] = self.paths[_token_in][_token_out]
```

But, because of the logic in `add_path()` described above, the first address in the obtained path could be either `_token_in` or `_token_out`. In the case that it is `_token_out`, the swap will fail as described below.

The stored path is passed along for use by the Uniswap V3 `SwapRouter` which assumes that the first 2 addresses in the path define the first pool to use. The `SwapRouter` uses Path.decodeFirstPool() to extract the first address, and it is the first return value thus interpreted as tokenIn for the swap:

```
(address tokenIn, address tokenOut, uint24 fee) = data.path.decodeFirstPool();
```

In the case that the first token address in the path is actually the intended output token for this swap, the Uniswap V3 `SwapRouter` will attempt to transfer the input amount of the output token and fail because it does not have any allowance to access the Unstoppable `SwapRouter` contract's balance of the output token.

Impact

The planned margin trading functionality will not work as intended when multihop swaps are required between the margin/debt token and the position token. Since such swaps will work in one direction but fail in the other, traders would either be unable to obtain the desired position or, in the worst case, be unable to exit the position and recover their margin funds. Note that in the latter case, liquidators will also be unable to force exit of over-leveraged positions.

Code Snippet

SwapRouter.add_path() at https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/blob/4153c3e67ccc080032ba0bbaffd9a0c56a573070/contracts/margin-dex/SwapRouter.vy#L130-L134

```
@external
def add_path(_token1: address, _token2: address, _path: Bytes[66]):
    assert msg.sender == self.admin, "unauthorized"
    self.paths[_token1][_token2] = _path
    self.paths[_token2][_token1] = _path
```

Tool used

Manual Review

Recommendation

Modify the `SwapRouter` `add_path()` function: remove the `_token1` and `_token2` parameters and instead infer them from the contents of `_path`. Make a reversed copy of the `_path` and store both versions in the paths hashmap with key ordering that matches the path being stored. Add Vault tests for multihop swaps.

Discussion

141345

This one will inevitably result in user loss, high severity might be more appropriate.

twicek

Escalate for 10 USDC. Assuming this is true: `transfer the input amount of the output token`. There is no allowance given for the output token, it will always revert as said in the report. If someone can find an example where it leads to loss of funds I agree that it should be high severity, otherwise it should be medium severity.

sherlock-admin2

> Escalate for 10 USDC. Assuming this is true: `transfer the input amount of the output token`. There is no allowance given for the output token, it will always revert as said in the report. If someone can find an example where it leads to loss of funds I agree that it should be high severity, otherwise it should be medium severity.

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.

141345

Recommendation: change the original judging, to medium

> Escalate for 10 USDC. Assuming this is true: `transfer the input amount of the output token`. There is no allowance given for the output token, it will always revert as said in the report. If someone can find an example where it leads to loss of funds I agree that it should be high severity, otherwise it should be medium severity.

The mentioned case is only 1 scenario. The possible impactful scenario is, token1 is transferred in. But when close the position, the reverse process will revert, causing user fund locked.

Then the admin can use add_path to rescue the funds by changing the path.

Based on the above, I think medium is appropriate.

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/12

hrishibhat

Result: Medium Unique
Considering this a valid medium based on the above comments

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- twicek: accepted

maarcweiss

Fixed by storing both paths from token 1 to token 2 and from token 2 to token 1. Lacking test coverage for this PR though

Issue M-8: `MarginDex::execute_limit_order` will always revert

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/111

Found by

Dug

Summary

All calls to the `MarginDex`'s `execute_limit_order` will always revert due the limit order being cleared in storage ahead of a call to `remove_limit_order`.

Vulnerability Detail

The `execute_limit_order` function includes the following logic...

The limit order is being set to `empty(LimitOrder)` in storage. Then a call to `self._remove_limit_order` is called for the same `_uid`.

`_remove_limit_order` has the following logic...

The newly-empty order is read from storage, and it's associated `account` is used fetch the `limit_order_uids`.

The issue is that this account is the zero address, which is the default value after the order was cleared in the preceding function. This will cause the `uids` array to be empty, and the loop will revert when `i == len(uids) - 1`.

Impact

This means that all calls to `execute_limit_order` will revert, and no limit orders can be executed, which is a major piece of functionality for the protocol.

Code Snippet

https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/MarginDex.vy#L568-L597

https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/MarginDex.vy#L619-L632

Tool used

Manual Review

Recommendation

`execute_limit_order` should not clear the limit order in storage before calling `remove_limit_order`. This will allow the `limit_order_uids` to be fetched correctly.

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/15

maarcweiss

Fixed by removing the code where the limit order was deleted:
`self.limit_orders[_uid] = empty(LimitOrder)`

Issue M-9: Users trying to reduce their positions with market orders will always revert

Source: https://github.com/sherlock-audit/2023-06-unstoppable-judging/issues/120

Found by

TheNaubit, Yuki, stopthecap

Summary

Reducing the users' positions with market orders will always revert due to a wrong value assigned to the `min_amount_out` variable when swapping part of the funds out.

Vulnerability Detail

In the `Vault` contract, there is the `reduce_position` function, which is called by the users to reduce their positions. When the function is called, the `min_amount_out` variable used by UniswapV3 as a slippage protection is calculated with the following code:

When the function receives a `min_amount_out` equal to 0, it means the user wants to reduce the position with a market order (for example, to reduce it right now). But in that case, some slippage is calculated to bring some protection to the swap. That protection is calculated with the function `_market_order_min_amount_out` which basically does the following:

The important part (for us) in that code is the call to `_quote_token_to_token`, which basically queries the price relation of both tokens and multiply it by the `_amount_in` var. But... if we check the original function (the one I wrote at the beginning of this section), the value passed as `_amount_in` to `_market_order_min_amount_out` is the `position_amount` value instead of the amount of the position to be reduced (contained in the var `_reduce_by_amount`).

And since in the `_swap` function we are swapping only the `_reduce_by_amount` amount and `position_amount` is always greater than `_reduce_by_amount`, the `_min_amount_out` value that we send in the `_swap` function will be always greater than the greatest amount out we could receive, making the `_swap` function to always revert thus making the `reduce_position` to revert also.

Impact

Users won't be able to reduce their positions using market orders, potentially making them to lose funds due to not being able to reduce their positions in the

right moment (since they should use non-market orders, which may not be right thing for their situation).

Code Snippet

https://github.com/sherlock-audit/2023-06-unstoppable/blob/main/unstoppable-dex-audit/contracts/margin-dex/Vault.vy#L309-L311

Tool used

Manual Review

Recommendation

Change the code like this:

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/unstoppable-dex-audit/pull/5

maarcweiss

Fixed by shifting the calculation from using the whole position `position.position_amount` to the actual amount the user is trying to reduce `_reduce_by_amount`