PUBLIC

Code Assessment

of the Unstoppable Margin Dex Smart Contracts

29 May, 2024

Produced for

by

Contents

1 Executive Summary

Dear Unstoppable team,

Thank you for trusting us to help Unstoppable with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Unstoppable Margin Dex according to Scope to support you in forming an opinion on their security risks. The Notes section contains information that users of the system should be aware of.

Unstoppable offers a margin trading platform that leverages the existing liquidity of decentralized exchanges (DEXs).

The most critical subjects covered in our audit are functional correctness, precision of arithmetic operations, and front-running.

Front-running protection has improved, as there was previously missing slippage protection, see Stop-Loss missing slippage protection. Functional correctness has improved, as swaps could previously fail on external markets, see Position Can Become Impossible to Close Due to Zero Swaps. Precision of arithmetic operations has been improved as there were previously rounding issues when providing liquidity, see Inflation Attack on Newly Added Tokens.

The general subjects covered are specification and gas efficiency.

The specification has improved, as the changes made during the fixes review process make the system more robust than it was previously, see Large Liquidations Can Fail. Gas efficiency has improved, as there were a large number of unecessary storage writes and reads in the margin dex contract, see Reading Unused Values from Storage in MarginDex.

In summary, we find that the codebase provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	5
• Code Corrected	5
Medium-Severity Findings	7
• Code Corrected	6
• Specification Changed	1
Low-Severity Findings	9
• Code Corrected	5
• Specification Changed	1
• Code Partially Corrected	2
• Risk Accepted	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Unstoppable Margin Dex repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	04 Feb 2024	78242c2b609ea48c0280ff636cc98a484c52793f	Initial Version
2	14 May 2024	1037f8b5114c84e94c33a6eaa48f6e77b7cdf1e6	First Fixes
3	27 May 2024	65a79a48c70f4e7b85b97090eb735563cbcb5015	Second Fixes
4	29 May 2024	6f836493a972690014502f98b959def76dd7c98f	Third Fixes

For the vyper smart contracts, the compiler version `0.3.10` was chosen.

The following contracts are in the scope of the review:

```
margin-dex:
    SwapRouter.vy
    Vault.vy
    MarginDex.vy
```

The review was conducted under the assumption that the system is deployed to the Arbitrum blockchain. The system was analyzed using the Arbitrum FIFO sequencer transaction ordering model. If it is deployed on another chain in the future, other transaction ordering assumptions may be necessary and should be carefully evaluated.

2.1.1 Excluded from scope

The following contracts are explicitly not in the scope of the review:

```
spot-dex:
    Dca.vy
    LimitOrders.vy
margin-dex:
    FeeConfiguration.vy
    InterestRateConfiguration.vy
    LiquidateWithBounty.vy
testing:
    MockERC20.vy
    MockOracleSwapRouter.vy
    MockOracle.vy
    MockSwapRouter.vy
    MockUniswapRouter.vy
    MockVault.vy
    MultiHop.vy
```

```
utils:
    Univ3Twap.sol
```

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Unstoppable offers a margin trading platform that leverages the existing liquidity of decentralized exchanges (DEXs). It supports advanced order types such as limit, stop-loss, and take-profit orders. Traders can take on leveraged positions by borrowing funds from LPs, which in turn receive interest payments. The borrowed funds can only be used in long or short positions, they cannot be withdrawn. This allows for an undercollateralized loan model. The system will initially be deployed to the Arbitrum blockchain.

2.2.1 Vault

The vault contract implements the logic for providing liquidity and trading on margin. All assets are stored in the Vault.

Asset-related operations, such as providing or removing liquidity and depositing or withdrawing margin, are conducted directly through the vault. Trading-related operations, including opening, reducing, or closing positions and liquidations, are executed via a whitelisted MarginDex contract.

Liquidity providers can provide *single-sided* liquidity by depositing *whitelisted* ERC-20 tokens into the vault. The tokens can then be borrowed by Traders to open leveraged positions. Interest is charged on the borrowed amount. The interest rate is calculated in an external InterestRateConfiguration contract based on the utilization rate, which is the ratio of borrowed tokens to the total amount of liquidity.

Liquidity Providers can provide tokens in two different tranches: They can deposit into the *Safety Module* or the *Liquidity Module*. The Safety Module is the junior tranche of the vault and is the first to cover protocol losses in case some liquidations create bad debt. Any losses surpassing the Safety Module are covered by the Liquidity Module. The Safety Module is compensated for its increased risk by receiving a higher interest rate (the percentage of interest it receives is configurable). For each Liquidity Provider, we store their pro rata share of the total amount of liquidity in the vault.

$$share = amount/index$$

When Traders close or reduce their position, they pay interest to the LPs, and the index increases. A portion of the interest is allocated to the Safety Module and the remainder to the Liquidity Module. When a Liquidity Provider withdraws their tokens from the vault, they receive their proportion of the total liquidity (incl. interest paid):

$$amount = share * index$$

Liquidity Providers must wait for a cooldown period (set by admin) after the last deposit before they can withdraw their tokens.

Traders deposit their initial margin in a *whitelisted* margin token to the vault. They can open leveraged positions by borrowing tokens from the vault and trading them for position tokens via the SwapRouter contract. The vault does not perform any access control on the trader and allows any whitelisted MarginDex contract to alter the trader's position.

The maximum quantity of tokens a trader can borrow is determined by their leverage ratio. The leverage ratio is defined as:

$$leverage = debtValue/(marginValue + positionValue - debtValue)$$

The value is the number of tokens times their USD oracle price. Currently, prices are provided only by Chainlink. The debt value is stored in shares and is calculated as:

$$debtvalue = debtshares * index$$

The interest on the debt increases each second:

$$index = index + (now - lastUpdate) * interestRate * debt$$

The interest is compounded each time the interest rate or debt is updated. An annual percentage rate of interest (APR) of 10% therefore adds up to 10.5% in one year if the contract is regularly called. Traders can reduce or close their positions. The profit or loss is realized, and the trader receives the remaining margin alongside the profit or loss in their margin token. The maximum leverage a user is allowed to have without being liquidated is configured per market by the admin. For example, if the maximum leverage of a market is 10, the user will be liquidated once their position reaches a leverage of 11 (10.99 is ok). Unstoppable supports margin tokens that are different from the position and debt token used. The allowed margin tokens per market are also configured by the admin. A separate maximum leverage can be set per margin token. If the market has a higher leverage limit than the margin token, the lower of the two limits will be used. Traders can increase or decrease the margin of their open position as long as their position remains healthy.

When a trader exceeds the maximum leverage, they can be liquidated. The liquidation process forcibly closes the position (using the oracle price for slippage protection) and potentially sells all margin tokens to cover the debt. If there is any margin left, the user pays a liquidation penalty that is taken by the protocol. Note that liquidations are not possible in case the Arbitrum sequencer is down.

If a user's margin and position tokens are not sufficient, the vault will have bad debt after closing the position. The loss is then covered by the Safety Module (and the Liquidity Module if necessary), by reducing the LP share price. If the bad debt of one token type becomes larger than its configured maximum acceptable value, the entire vault (all token types) automatically goes into "defensive mode" until the administrator disables the mode. In defensive mode no new positions can be opened and no funds can be added to the system, but users can still close positions and withdraw. Bad debt of the system can be paid back by anyone, although there is no incentive to do so. This would most likely only be done by an insurance fund that is funded from fees taken by the protocol. When bad debt is paid back, the LP share price that was previously reduced is increased again.

2.2.2 SwapRouter

The Swap Router routes all trades via Uniswap V3. It supports direct routes and indirect routes routing through a base pool. The admin can add/remove new trading pairs to the router. In Version 1, the SwapRouter to be used was updated by the Vault admin. In Version 2, it is set by the MarginDex admin. In the future, a SwapRouter that routes through additional DEXes, not just Uniswap, could be added.

2.2.3 MarginDex

The Margin Dex checks whether the senders have access rights to create or alter positions. Users can delegate to other addresses and give them the ability to act on their behalf. The delegate will be able to trade for the user, but not withdraw any funds from the user's vault margin balance.

To create a position, users can either directly call the `open_position` function or create a limit order. A limit order defines the `amountIn` and the `minimumOutReceived` to ensure that the order can only be executed if they receive the trade at the specified exchange rate or better. The limit order can only be executed if the user has a sufficient amount of margin available in the vault. If the user creates multiple orders using the same margin token and there are not enough margin tokens to cover all, only the first order will execute. However, if an additional margin is added later, the other limit orders may also become executable. See also: Limit Orders can Become Executable below Market Price. The execution of orders is permissionless, meaning any party can call the contract to execute the order.

Users can set *stop-loss* and *take-profit* orders to close or reduce an open position. A stop-loss order can only be executed if the Chainlink price drops below a specified price threshold. The take profit order enforces the `amountIn` and `minimumOutReceived`, so it can only be executed at the specified

exchange rate or better. The execution of both types of orders is permissionless and can be executed by any party. Take-profit orders and stop-loss orders remove some margin proportionally to the change of the position size: Reducing the position by x% will reduce the margin by x% as well.

Liquidations must be conducted through the Margin Dex. The account calling the `liquidate()` function does not receive any explicit reward for this. It is expected that the "Liquidation Engine" run by Unstoppable off-chain will call the permissionless functions such as `liquidate()` and `execute_limit_order()` within a short time frame whenever they hit their execution conditions if nobody else does. However, in many cases, there is an incentive for other entities to call these functions and profit from them through arbitrage. See also: Sandwiching order execution.

The admin can set `is_accepting_new_orders` to false, which makes it impossible to create or execute orders. However, liquidations can still happen. The admin can also set the vault address. If this address is not set correctly, no orders and no liquidations can be executed.

2.2.4 Trust Model

Delegates are fully trusted by the user delegating to them and have the power to modify the user's positions and initiate trades on their behalf, including setting their slippage parameters. In the worst case, the delegate could steal funds from the user by creating trades with high slippage and sandwiching them.

The InterestRateConfiguration is partially trusted. It is expected to return an appropriate interest rate. In the worst case, it could return very high interest rates, that can make traders with open positions pay all their margin to LPs and then liquidate them. It could also cause a DoS to the system if it returns values that are close to `uint256.max`, as `_update_debt()` would revert.

The admin of the Vault is fully trusted. It can modify critical admin parameters including the interest rate, the slippage limits, the maximum leverage, the trading pairs, pausing new orders, whitelisting new MarginDex instances, and onboarding new tokens. In the worst case, the admin can steal all funds in the system by setting malicious oracles and liquidating positions with bad debt, or by enabling bad tokens as margin.

The admin of the MarginDex is partially trusted. In the worst case, they can cause a DoS, making it impossible to open/close positions or liquidate them, until the Vault admin replaces the MarginDex contract. Not being able to liquidate can cause bad debt, which would result in a loss for LPs.

The price oracle (initially only Chainlink is supported) is trusted to provide correct and up-to-date prices. In the worst case, an incorrect oracle price could lead to swaps with very high slippage, potentially leading to bad debt, or to a DoS, where all swaps using the system slippage (such as liquidations) revert.

The Arbitrum sequencer is assumed not to go down for extended periods of time. If it does, liquidations will be impossible, which can lead to bad debt for the system. See also: Behavior in Case of Sequencer Downtime.

The off-chain "Liquidation Engine" is assumed to execute orders, and especially liquidations, as soon as possible. To do this, it should have a robust RPC setup, which continues functioning even when the chain is under high load.

2.2.5 Changes in Version 2

The following changes were made to the specification in **Version 2** of the codebase:

- Trades can now be routed through any contract that supports the `P2PSwapper` interface, not just through the SwapRouter. When opening a position, the `P2PSwapper` to be used must be the MarginDex. However, when adjusting or closing a position, the caller can specify their own `P2PSwapper` to be used. This allows routing trades through any market, which should improve available liquidity. The price of the trade must still be "fair", meaning it cannot exceed the configured slippage compared to the price oracle. For trades executed by the user themselves (not a liquidation, take-profit, or stop-loss), the allowed slippage limit is twice as

large. Any implementation of `P2PSwapper` must ensure it either does not hold tokens, or does not have open approvals to the vault in between calls.

- An explicit liquidation bonus, on top of any extra price impact captured by the liquidator, has been added to incentivize quick liquidations.
- Partial liquidations are now possible.
- The SwapRouter is now set by the MarginDex admin, not the Vault admin.
- A liquidator whitelist has been added. Trades that result in bad debt for the protocol (according to the oracle) can only be executed by addresses on the whitelist. Liquidations that do not result in bad debt are still permissionless.

2.2.6 Changes in Version 3

The following changes were made to the specification in **Version 3** of the codebase:

- A caller whitelist was added to MarginDex. Only addresses that opt-in to the whitelist can now be used for the `P2PSwapper.flash_callback` call. This was added in response to Users With Approvals Can Be drained.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation
- Trust: Violations to the least privilege principle

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3

- Flash_callback Can Return Incorrect Values Risk Accepted
- Debt Is Rounded in Favor of User Code Partially Corrected
- Reading Unused Values From Storage in MarginDex Code Partially Corrected

5.1 Flash_callback Can Return Incorrect Values

Correctness Low Version 2 Risk Accepted

CS-UNM-031

In MarginDex, the `flash_callback` function returns the `actual_outs` array, which lets the vault know how many tokens were received, so that it can transfer them back.

For `exact_in` orders, the return value is calculated as follows:

```
if swap_type == 0: # exact_in
    actual_out: uint256 = self._swap(token_in, token_out, amount_in, amount_out)
    actual_outs[s[1]] += actual_out
    if actual_outs[s[0]] > 0: #if we have received some token_in from a previous swap
        if actual_outs[s[0]] >= amount_in: #if we have enough previous in to cover the trade
            actual_outs[s[0]] -= amount_in
        else:
            actual_outs[s[0]] = 0 #if we don't have enough previous in to cover the trader
```

In the last line, we set `actual_outs[s[0]] = 0` if the previous swap did not give us enough tokens to cover the full trade. However, we may have also transferred some of the same token into the MarginDex from the Vault. Any remainder from the transferred in amount will not be reflected in the return value. As a result, some tokens could be remaining in the MarginDex that should have been transferred back to the Vault. These tokens can be claimed by anyone.

For `exact_actual_out` orders, the return value is calculated as follows:

```
if swap_type == 2: # exact_actual_out
        remaining_needed: uint256 = amount_out - ERC20(token_out).balanceOf(self)
```

```
        if remaining_needed > 0:
            actual_in: uint256 = self._swap_exact_out(token_in, token_out, remaining_needed, amount_in)
            actual_outs[s[1]] += remaining_needed
            actual_outs[s[0]] += amount_in-actual_in
        else: #no trade
            actual_outs[s[0]] += amount_in
```

In the last line, any amount that was transferred from the vault is meant to be accounted, so that it is correctly transferred back. However, the `amount_in` value can also be from a previous trade. In this case, the amount was already added in the previous trade. As a result, the same tokens will be added to the `actual_outs` twice. When the Vault tries to `transferFrom()` this amount, the balance will be insufficient and the transaction will revert. However, the user can try again with a different swap_sequence and succeed.

Risk accepted:

Unstoppable is aware of the issue and accepts the risk of incorrect return values.

Unstoppable responded:

```
Non issue for real world swap sequences.
```

5.2 Debt Is Rounded in Favor of User

Design | Low | Version 1 | Code Partially Corrected

CS-UNM-011

The function `_amount_per_debt_share` rounds the amount per debt share down, causing `_amount_to_debt_shares` to round up and `_debt_shares_to_amount` to round down. This rounding benefits the user in several functions:

1. `reduce_position` rounds up `shares_to_burn`.
2. `_close_position` rounds down `position_debt_amount`.
3. `liquidate` rounds down `position_debt_amount`.
4. `_effective_leverage` rounds down `debt_value`.

It is considered best practice to never round in favor of the user and always round in favor of the protocol.

Code partially corrected:

The rounding has been adjusted in favor of the protocol where possible. In `change_position`, during partial liquidations, debt is still rounded in favor of the user.

Unstoppable responded:

```
The only remaining rounding issue we see is that during partial closes the rounding is “in favour”
of the user but at the expense of other debtors (including the remaining debt of that same user).
This plus the fact that rounding occurs at an extra 18 decimals leads us to assume this is not an issue in our case.
Any attempt to adjust for this lead to serious issues in the internal accounting, “negative” debt etc.
```

5.3 Reading Unused Values From Storage in MarginDex

Design | Low | Version 1 | Code Partially Corrected

CS-UNM-012

The MarginDex reads values that are never used from storage and writes values that are unchanged back to storage. This causes unnecessary gas consumption.

1. **User IDs:**

The open trades and limit orders IDs of each user are stored in an array of 1024 elements. In `_cleanup_trade()` and `_remove_limit_order()`, the entire array is copied from storage to memory, an element is removed, and then the modified array is written back to storage. By operating on the array in storage instead of copying it to memory, it would be possible to reduce the (average) number of storage reads by half and write to storage only 2 times.

2. **Large Order structs:**

Trades and limit orders are stored in large structs (>1.6 kb). The full structs are read from storage in the following functions, even if only a few elements are used.

Trade

1. *close_trade*
2. *partial_close_trade*
3. *update_tp_sl_orders*
4. *add_margin*
5. *remove_margin*
6. *execute_tp_order*
7. *execute_sl_order*
8. *cancel_tp_order*
9. *cancel_sl_order*

LimitOrder

1. *cancel_limit_order*

Only accessing (and writing back) the elements of the struct that are used would significantly reduce the gas cost of these functions.

Code partially corrected:

The code has been changed in some but not all places to improve gas efficiency.

Unstoppable responded:

```
In some instances we prefer clean & readable code to minimized gas costs, especially since we're on an L2.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	5

- Bad Debt Can Be Faked for Profit Code Corrected
- Users With Approvals Can Be Drained Code Corrected
- Large Liquidations Can Fail Code Corrected
- Position Can Become Impossible to Close Due to Zero Swaps Code Corrected
- Stop-Loss Missing Slippage Protection Code Corrected

Medium-Severity Findings	7

- Bad Debt Check Is Ineffective Code Corrected
- Swap Margin Uses Incorrect Fairness Check Code Corrected
- TP / SL Can Increase Debt Exposure Code Corrected
- Amount Returned From SwapRouter Is Not Validated Code Corrected
- Inflation Attack on Newly Added Tokens Code Corrected
- MarginDex Admin Is More Trusted Than Required Code Corrected
- Stop-Loss Can Unintentionally Increase Leverage Specification Changed

Low-Severity Findings	6

- Reentrancy Into flash_callback Code Corrected
- Bad Debt Check Is Inaccurate Code Corrected
- Blacklisted Tokens Can Be Swapped Into Code Corrected
- Close_position Slippage May Be Too Strict Code Corrected
- Multiple IDs for Each Position Code Corrected
- Vault Assumes Chainlink Oracles Have 8 Decimals Specification Changed

Informational Findings	4

- Floating Pragma Code Corrected
- Event Logs Value With Unclear Interpretation Code Corrected
- Vault Uses Incorrect ERC20 Function Interface Code Corrected
- SwapRouter Admin Cannot Be Changed Code Corrected

6.1 Bad Debt Can Be Faked for Profit

Correctness High Version 2 Code Corrected

CS-UNM-023

In Vault, the `change_position` function checks that the `position.margin_amount` is zero before realizing bad debt.

```
if position.position_amount == 0:
    # no position left -> full close
    if position.debt_shares > 0:
        # bad debt case
        assert self.is_whitelisted_liquidator[_caller] or self.bad_debt_liquidations_allowed,
        "only whitelisted liquidators allowed to create bad debt"
        assert position.margin_amount == 0, "use margin to lower bad_debt"
```

However, the `position.margin_amount` is updated later in the function, after this check was made:

```
if actual_margin_in > 0:
    self._safe_transfer_from(position.margin_token, _caller, self, actual_margin_in)
    position.margin_amount += actual_margin_in
```

A user could liquidate themselves and trade all of their token value into `margin_amount`. This would pass the "fairness check", as the trade would clear at a fair price. Then the user could have their debt deleted and accounted for as bad debt, as their position margin is zero at the time of the check. Afterward, they will receive the `margin_amount` from the trade back. This would allow them to profit from the bad debt.

However, this is only possible if the `bad_debt_liquidations_allowed` flag is set to true, or if the user is on the liquidator whitelist. If one of these conditions hold, all funds in the contract would be at risk.

Code corrected:

The `position.margin_amount` is now updated before the check for bad debt. This resolves the issue.

6.2 Users With Approvals Can Be Drained

Security | High | Version 2 | Code Corrected

CS-UNM-027

In the `flash_callback` functionality of the Vault, any address can be provided as the `_caller`, which will be used to swap tokens. In particular, any user's EOA can be used as a "Swapper". The provided address will receive some tokens, then `transferFrom()` will be called to transfer the received tokens back to the Vault.

If a user has an open approval to the Vault, for example, because they are about to make a deposit, then all their approved tokens can be taken by any trader, at a price of zero. This passes the fairness check, as it is a very good price for the trader.

If all users always deposit to the system using a multicall that approves tokens and deposits them in a single transaction, and never give more approval than required, then they are safe. The default Unstoppable frontend uses such a multicall flow for deposits. However, users typically expect it to be safe to give approval to a contract they trust.

Sophisticated users, that intentionally implement the `P2PSwapper` interface, must ensure that they never have open approvals between calls.

Code corrected:

A `is_whitelisted_caller` whitelist has been added to the MarginDex. Now, addresses need to opt-in to the whitelist to be allowed as `flash_callback` targets. This ensures that normal users cannot

be used as swappers, unless they call the `set_is_whitelisted_caller` function and intentionally add themselves to the whitelist.

Contracts that add themselves to the whitelist must still ensure that they never have open approvals to the vault between calls, unless they do not hold any tokens.

If a new MarginDex is added to the Vault in the future, it should be ensured that it also has a similar whitelist mechanism.

6.3 Large Liquidations Can Fail

Design High Version 1 Code Corrected

CS-UNM-001

Positions in Unstoppable Margin Dex can only be fully liquidated, never partially. Additionally, liquidations are always done with a Uniswap swap, using a `min_amount_out` based on the oracle price and the `liquidate_slippage` set by the admin.

If there is a very large position, that causes more than `liquidate_slippage` of slippage on Arbitrum Uniswap when attempting to close it in a single swap, this swap will revert. As a consequence, it will be impossible to liquidate the position unless the liquidity on Uniswap improves or the `liquidate_slippage` is increased. Delaying a liquidation is a risk to the system and can cause bad debt.

If functionality was added to partially liquidate a position, the large position could be liquidated in chunks, where each chunk would have a smaller slippage on Uniswap than liquidating the full position at once. This would reduce the chances of a liquidation being impossible due to the `liquidate_slippage` being too small. Note that if the oracle price is higher than the Arbitrum Uniswap price, it still might not be possible to execute even partial liquidations.

Code corrected:

In Version 2, liquidators can partially liquidate the position amount, but not the margin amount. When the margin amount is sold, the function `Vault.change_position()` expects a full liquidation and requires repayment of the entire debt and position. If there is a position with a large margin and debt amount but a small position amount (e.g. 1 wei), the full liquidation may cause more slippage than the fairness check allows, and the liquidation reverts.

In Version 3, it is allowed to trade from margin during a partial liquidation. This also allows partial liquidations of positions that have a large amount of margin and debt left as long as some position is left.

6.4 Position Can Become Impossible to Close Due to Zero Swaps

Correctness High Version 1 Code Corrected

CS-UNM-002

The `_swap` function in the Vault attempts to perform a swap on an external DEX for all values passed (including zero). The only DEX that is currently supported is Uniswap V3. Uniswap V3 reverts when the specified amount is zero: https://github.com/Uniswap/v3-core/blob/d8b1c635c275d2a9450bd6a78f3fa2484fef73eb/contracts/UniswapV3Pool.sol#L603

The vault calls `_swap()` with amount = 0 when a user's position is closed and one of the following cases applies:

1. The user has reduced their position amount to 0 using a partial close

2. The user removed their full margin amount and marginToken is not the debtToken
3. Trader PnL is 0 and marginToken is not the debtToken

In all these cases, `close_position()` will revert, and it will be impossible to close the position.

An attacker can abuse this behavior, since it stops them from getting liquidated. For example, an attacker who has a sufficiently high PnL could remove all margin from his position, to avoid liquidation. Note that they can always deposit some margin tokens back, to close their position. This lets them take all the potential upside of a trade, while avoiding the downside.

Note that the Vault Admin could make the position closeable again by replacing the SwapRouter with a new one that returns immediately in case the swap amount is 0.

Other decentralized exchanges may revert for other values (e.g. Curve's CryptoSwap Newton Algorithm only converges for values in a certain range).

Code Corrected:

The vault no longer directly calls the `_swap()` function on the SwapRouter. Instead, it calls the `P2PSwapper` interface's `flash_callback()` function. The `flash_callback()` implementation in MarginDex handles zero amounts without reverting.

6.5 Stop-Loss Missing Slippage Protection

Design | High | Version 1 | Code Corrected

CS-UNM-003

The `execute_sl_order` function takes a caller-provided `_min_amount_out` argument. However, the function can be called by anyone. This means the `_min_amount_out` can be set arbitrarily low, effectively rendering it useless.

An attacker can profit from this by sandwiching the call. The following calls can be executed atomically in a single transaction:

1. Move the Uniswap Arbitrum market by selling the position asset
2. Call `execute_sl_order()`, selling position tokens with slippage
3. Buy the position tokens on Uniswap at a discount and bring the market price back to the initial value

The attack has no capital requirements, as it can be executed using a flashloan. It is profitable if the trading fees to move the market are lower than the profit from the sandwich.

The maximum amount of slippage that can be incurred is dependent on the type of Stop-Loss order (full or partial close) and the leverage ratio of the position. For full closes, the maximum slippage allowed by the system for liquidations is the limiting factor. For partial closes, this limit is not in place. However, the position may not get into a liquidatable state after partially closing and withdrawing part of the margin.

Consider the following example of a partial close on a pair where the liquidation threshold is 8x leverage and positionToken/debtToken oracle price is 1:

1. A user has a position of size 1000 with debt 1000 and margin 500 (2x leverage)
2. A Stop-Loss with size 500 reaches its trigger price
3. The user's Stop-Loss is executed by someone who sandwiches them to the maximum extent
4. The execution price of the Stop-Loss is 2/3 (33% slippage), leaving the user with position size 500, debt (1000 - 500*2/3) = 666.66 and margin 500
5. 250 margin is withdrawn, leaving the user at 7.99x leverage, on the edge of liquidation

The user incurred a slippage loss of 500/3 = 166.66 tokens. This amount will be a profit to the sandwicher, which will then need to deduct the costs of executing the sandwich. Users that start with lower leverage will incur more slippage than those that are already near the leverage limit. If a user has positive PnL at the stop loss trigger price, they will also lose their PnL amount to slippage.

The `_min_amount_out` for Stop-Loss execution should be lower-bounded by a value provided by the system or the user.

Code corrected:

The maximum price impact of trades is now always limited by the fairness check in `Vault.change_position()`.

6.6 Bad Debt Check Is Ineffective

Design | Medium | Version 2 | Code Corrected

CS-UNM-028

In `Vault.close_position()` only whitelisted liquidators should be allowed to create bad debt as long as `bad_debt_liquidations_allowed` is set to `False`. However, the check is ineffective as it is only enforced when a position is fully closed.

```
if position.position_amount == 0:
# no position left -> full close
        if position.debt_shares > 0:
            # bad debt case
            assert self.is_whitelisted_liquidator[_caller] or self.bad_debt_liquidations_allowed,
            "only whitelisted liquidators allowed to create bad debt"
```

A liquidator that is not whitelisted can still partially close a position by selling all but 1 wei of the position amount. This creates a position with *unrealized bad debt*:

$$positionValue + marginValue <= debt$$

Code corrected:

A check has been added that ensures that the positionValue + marginValue is greater than the debt after a position is changed by a non-whitelisted liquidator. This means that there cannot be any unrealized bad debt in this case, given that the oracle price used to calculate these values is correct.

6.7 Swap Margin Uses Incorrect Fairness Check

Correctness | Medium | Version 2 | Code Corrected

CS-UNM-029

Function `Vault.swap_margin()` defines a fairness check to ensure that the user sets an appropriate minimum amount they should receive from the swap ("amount1"). However, the implemented check does not enforce a lower bound on the amount received but instead ensures that the amount returned is lower than the amount expected by the user.

```
assert (self._quote(_token0, _token1, _amount0) * (PERCENTAGE_BASE + self.reasonable_price_impact[_token0])
/ PERCENTAGE_BASE) >= _amount1, "unfair margin swap"
```

The code later asserts that the realized amount is greater than the amount1, allowing the user to get a better price than the oracle price:

```
assert actual_amount1 >= amount1, "[MS:cb] too little out"
```

However, there is no guarantee that the user will receive at least the oracle price minus some slippage, as intended.

$$inAmount >= outAmount * (1 + slippage)$$

As `MarginDex` limits access to this function to the user themselves and their delegatee, (who is fully trusted) the impact is limited.

Code corrected:

The fairness check has been corrected to ensure that the user receives at least the oracle price minus some slippage.

6.8 TP / SL Can Increase Debt Exposure

Design | Medium | Version 2 | Code Corrected

CS-UNM-030

In Version 2 of the MarginDex contract, the `execute_tp_order` and `execute_sl_order` functions allow the caller to set arbitrary values for `_debt_change`, `_margin_change`, and `_realized_pnl`. A malicious actor could choose `_debt_change < 0` and `_realized_pnl > 0` to create additional debt and offset the debt by moving more margin to the position. This is permissible as long as the final position meets the leverage criteria and the trade is fair.

For instance, consider a trader with an initial position of:

- position_amount = 100
- margin_amount = 10
- debt_amount = 100
- leverage = 10
- trader margin balance = 5

A malicious executor could execute a take-profit (or stop-loss) order with the following parameters:

- reduce_by_amount = position_change = -50
- debt_change = -50
- margin_change = +100
- realized_pnl = +5

The trade can be executed since the fairness condition is fulfilled:

$$positionChange + debtChange = marginChange$$

After the trade, the trader has a modified position that has the same leverage as before, but higher debt:

- position_amount = 50
- margin_amount = 115
- debt_amount = 150
- leverage = 10
- trader margin balance = 0

The attacker profits from executing the orders, as higher trading volume increases the extractable slippage.

Further, users with standing TP/SL orders can be griefed by executing their orders with a negative `realized_pnl` to consume all of their available margin. This stops them from opening a new position, as they are unable to pay the trading fee.

Code corrected:

The following checks have been added to the `execute_tp_order` and `execute_sl_order` functions:

```
assert _debt_change >= 0, "no new debt during tp/sl"
assert _realized_pnl <= 0, "cannot add more margin during tp/sl"
```

For TP/SL orders that do not fully close a position, additional checks are made:

```
assert not is_full_close, "was supposed to be full close"

assert Vault(self.vault).positions(_trade_uid).margin_amount
<= position.margin_amount, "cannot increase margin on partial tp"
```

This ensures that no unexpected changes to the position can be made.

6.9 Amount Returned From SwapRouter Is Not Validated

Correctness **Medium** **Version 1** **Code Corrected**

CS-UNM-004

According to the trust model of the system, the SwapRouter is not fully trusted. As such, the Vault verifies that it has received the expected quantity of tokens from the swap by comparing the token balance before and after the swap.

```
token_out_balance_before: uint256 = ERC20(_token_out).balanceOf(self)

amount_out_received: uint256 = SwapRouter(self.swap_router).swap(
    _token_in, _token_out, _amount_in, _min_amount_out
)

token_out_balance_after: uint256 = ERC20(_token_out).balanceOf(self)
assert (
    token_out_balance_after >= token_out_balance_before + _min_amount_out
)
```

However, the vault only verifies that the token balance has increased by at least `min_amount_out`, without confirming it has increased by `amount_out_received`.

The received amount is used to calculate the Trader's profit. If the SwapRouter returns a value that is too large, this extra amount will be credited to the trader, which can then be withdrawn. In the worst case, this could drain all funds in the vault and make it insolvent.

Code corrected:

The Vault no longer directly calls the SwapRouter. Instead, it calls an untrusted `P2PSwapper` contract. The value returned by the `P2PSwapper` is checked to be at least as large as expected. Afterward, a `safe_transfer_from` call is used to transfer this amount of tokens from the `P2PSwapper` to the Vault. If the `P2PSwapper` does not have a sufficient balance, the transfer will revert. As a result, an incorrect return value can no longer cause issues.

6.10 Inflation Attack on Newly Added Tokens

Security | Medium | Version 1 | Code Corrected

CS-UNM-005

Shared Vaults such as the Unstoppable Vault have a known issue called an "inflation attack", where an attacker increases the price per share of vault shares by a lot, such that a small rounding error in the number of shares other users receive results in a large percentage of the deposit being lost to rounding. This is generally only possible if the attacker is the first depositor to an empty vault.

The Unstoppable Vault mitigates this attack in 2 key ways:

1. It does not count any tokens sent directly to the vault as belonging to LPs, so it is hard to "donate" to the LP.
2. When minting initial shares, `10E18` shares are minted per wei of token deposited. This makes the total number of shares large, even if only `1` wei is deposited. Additionally, withdrawal amounts can only be specified in underlying tokens, not shares.

However, both of these mitigations can be circumvented:

1. Tokens cannot be donated to the LPs directly, but interest from borrowing can accrue and be paid to LPs. In particular, interest from funds lent by one module, such as the `base_lp`, can be paid to the other module, such as the `safety_module_lp` (or the other way around), even if the `safety_module_lp` has no shares or few shares. This allows donating a lot of interest (by depositing a lot of `base_lp` and borrowing it all) while keeping the number of `safety_module_lp` shares small.
2. Once the price per share has been increased by at least `1E18`, it will be possible to withdraw a partial amount of the initially minted shares and only leave `1` share left.

Combining these factors, the following attack can be executed on any token that is newly added to the system and has no LPs in one of the modules (here illustrated with the `safety_module_lp`) yet:

1. Deposit a large amount to the `base_lp`
2. Borrow 100% of available liquidity from yourself.
3. Deposit `1` wei of token to `safety_module_lp`. Receive `1E18-1` shares.
4. Partially close an amount of the borrow position such that exactly `1E18-2` tokens are paid as fees to the `safety_module_lp`. Now the price per share is `1` (initially it was `1/(1E18-1)`). This allows us to withdraw a precise number of shares.
5. Withdraw `1E18-2` tokens from the `safety_module_lp`. Now there is `1` wei token and `1` wei shares left.
6. Wait for a significant amount of interest to accrue. Close the borrow position. If `X` interest accrued to the `safety_module_lp`, the price per share of `safety_module_lp` will now be `X+1`.
7. Wait for another user to deposit to the `safety_module_lp`. When they do, the shares minted will be rounded up, then reduced by `1`. This means they will receive up to `1` share less than if there was no rounding. Since `1` share is worth `X+1` wei tokens, up to `X+1` wei tokens will be accounted as belonging to the existing LPs (the attacker) instead of the depositing user.

Note that all future users will suffer from up to `X+1` rounding, not just the first one:

- In step 6., the attacker pays interest of `1000E18` USDC to the `safety_module_lp`
- As there is only one share, the price per share is now `1000E18 + 1`.
- A second user deposits `2000E18` USDC in step 7. The rounded-up number of shares they should receive is `2`. This number is reduced by `1` to account for rounding. The user only receives `1` share and the price per share now increases to `1500E18`. The attacker made a profit of `500E18` USDC, which came from the second user.
- A third user deposits `1500E18` USDC. Their rounded-up shares are `1`, which is reduced by `1`, so they receive `0` shares. The price per share increases to `2250E18`. The attacker and the second user both make `750E18` USDC profit, which came from the third user.

The maximum amount that can be donated in step 6. is limited by the following factors: The amount the attacker deposits, the maximum interest rate, the percentage of interest that goes to the `safety_module_lp`, and the time deposited. The time deposited is likely the largest limiting factor, as the attacker only has time until the first user deposits, after which the attacker will no longer own all the shares in the pool.

In summary, the mitigations in place for inflation attacks are insufficient, and, given enough capital and time, an attacker can create a pool where all future users lose 100% of their deposit.

Code corrected:

The inflation attack was mitigated by adding a virtual offset to the `_amount_to_lp_shares` and `_lp_shares_to_amount` functions.

For details on this approach, see: ERC4626 - defending with a virtual offset.

6.11 MarginDex Admin Is More Trusted Than Required

Trust | Medium | Version 1 | Code Corrected

CS-UNM-006

The MarginDex admin is a partially trusted role. It should be able to set the MarginDex instance's `is_accepting_new_orders` flag, which pauses new orders but still allows existing positions to be closed.

However, the admin can also set the `vault` variable, which points to the Vault on which orders are executed. If the admin sets this variable to a different address, it will be impossible to interact with the Vault, unless another MarginDex contract is added by the Vault admin. This would cause a Denial of Service, making it impossible to adjust positions or liquidate them.

The `vault` variable doesn't need to be updatable, so trusting the MarginDex admin not to maliciously unset the Vault is an unnecessary risk.

Code corrected:

The MarginDex admin can no longer set the `vault` variable. The `vault` variable is now set in the constructor and cannot be changed.

6.12 Stop-Loss Can Unintentionally Increase Leverage

Design | Medium | Version 1 | Specification Changed

CS-UNM-007

In MarginDex, users can specify a Stop-Loss order that reduces their position at a certain price. If it is a partial close, some of the user's margin will also be withdrawn when the order is triggered.

This is done as follows:

```
amount_out_received = self._partial_close(trade, sl_order.reduce_by_amount, _min_amount_out)
ratio: uint256 = sl_order.reduce_by_amount * PRECISION / position.position_amount
remove_margin_amount: uint256 = position.margin_amount * ratio / PRECISION
Vault(self.vault).remove_margin(trade.vault_position_uid, remove_margin_amount)
```

The amount removed from margin is the percentage of margin that corresponds to the percentage of the position that was partially closed. E.g. if the position was reduced by 10%, the margin is also reduced by 10%. This is intended to keep the leverage of the position unchanged.

However, the `ratio` calculated here assumes that the partial close reduces the debt by the same percentage that the position is reduced. This is only the case if the trade executes exactly at the oracle price which is used to calculate the leverage.

This may not happen for multiple reasons:

1. The oracle price may not reflect the current price
2. There may be trading fees
3. There may be slippage

If the execution price is below the oracle price, the debt of the position will be reduced by less than `ratio`. Withdrawing `ratio` of the margin will thus unintentionally increase the leverage of the position. In the extreme case where the position already has high leverage, this may cause the leverage to increase above the maximum acceptable leverage. The transaction will revert and it will be impossible to execute the partial-close Stop-Loss, even though it would have been possible if less margin was withdrawn.

The same issue also affects Take-Profit orders.

Specification changed:

The specification has been updated to allow changes in leverage up to the bounds set by the `leverage_buffer` parameter. This ensures that the leverage does not increase too much:

```
if Vault(self.vault).positions(_trade_uid).position_amount > 0:
    # partial close
    leverage_after: uint256 = self._effective_leverage(_trade_uid, 2)
    assert (leverage_after >= leverage_before - min(leverage_before,
    self.leverage_buffer)) and (leverage_after <= leverage_before + self.leverage_buffer),
    invalid sl execution"
```

6.13 Reentrancy Into `flash_callback`

Design | Low | Version 2 | Code Corrected

CS-UNM-032

The function `flash_callback` in the `MarginDex` contract is assumed to be called only by the Vault contract. However, there is no restriction on calling it from another contract, and no protection against reentrancy. A malicious decentralized exchange or tokens with callbacks could reenter the `flash_callback` function. As the margin dex holds funds for the duration of the trade, a part of the funds could be stolen in this situation.

The maximum value that can be extracted is capped, as the amount of tokens returned must still fulfill the fairness criteria enforced by the `Vault` contract.

Code corrected:

A reentrancy lock was added to the `flash_callback` function. It is a separate lock from the one used in other functions of MarginDex.

6.14 Bad Debt Check Is Inaccurate

Correctness | Low | Version 1 | Code Corrected

CS-UNM-008

In `close_position()`, there is a check that should ensure a user cannot close a position that would result in bad debt:

```
margin_value: uint256 = self._quote_token_to_token([...])
assert min_amount_out + margin_value >= self._debt(_position_uid), "min_amount_out cannot result in bad debt"
```

This check is inaccurate, as the `margin_value` is calculated using the oracle price. The actual price that swapping the margin token to debt token would execute at may be lower due to slippage. For the position token, this is taken into account by using the `min_amount_out`.

As a result, it may be possible for the user to close a position that results in bad debt.

Code corrected:

The `close_position` function was removed and replaced with the more generic `change_position` function, which checks a position's health based on the remaining tokens after the swap has taken place. While the oracle price is still used for the remaining tokens, the realized price of the swap is token into account for the swapped tokens.

6.15 Blacklisted Tokens Can Be Swapped Into

Design | Low | Version 1 | Code Corrected

CS-UNM-009

The `Vault` prevents the user from funding their account with assets that are not on the whitelist. However, they can call the `swap_margin` function to swap into tokens that were removed from the whitelist, given that they still have a route configured in the SwapRouter. They can then use the blacklisted token to top-up an existing margin position via `add_margin`.

Code corrected:

The `swap_margin` function now enforces a whitelist on the token being swapped into.

6.16 Close_position Slippage May Be Too Strict

Design | Low | Version 1 | Code Corrected

CS-UNM-010

In Vault, `close_position()` enforces that the `min_amount_out` given by the user is at least as large as the `system_min_out`.

```
assert min_amount_out >= system_min_out, "too little min_amount_out"
```

This condition is stricter than necessary. As there is a check later that the `min_amount_out` chosen cannot cause bad debt, a user should be allowed to specify a `min_amount_out` that is smaller than the one set by the system. A large value set by the system can cause the close to revert. This would be most likely to be a problem when the oracle price is higher than the Arbitrum Uniswap price.

Code corrected:

The `close_position` function was removed and replaced with the more generic `change_position` function. Trades are now protected from too much slippage by a single "fairness check" in `change_position`.

6.17 Multiple IDs for Each Position

Design | Low | Version 1 | Code Corrected

CS-UNM-037

The MarginDex stores two IDs for each position, a `position_uid` and a `vault_position_uid`. In the contracts in scope, both IDs are always the same.

Code corrected:

The secondary ID was removed, and a single `position_uid` is now used everywhere.

6.18 Vault Assumes Chainlink Oracles Have 8 Decimals

Correctness | Low | Version 1 | Specification Changed

CS-UNM-013

In Vault._to_usd_oracle_price, it is assumed that Chainlink USD price feeds have 8 decimals. This condition is not validated when a new asset and its price feed are whitelisted, so it is possible to add a price feed with a differing number of decimals.

Note that some price feeds like AMPL / USD have 18 decimals https://etherscan.io/address/0xe20CA8D7546932360e37E9D72c1a47334af57706#readContract

---

Specification Changed:

Unstoppable clarified that they plan on only utilizing price feeds from the "verified" tier with 8 decimals of precision.

6.19 Event Logs Value With Unclear Interpretation

Informational | Version 1 | Code Corrected

CS-UNM-017

The MarginDex emits the LimitOrderPosted event for each limit order placed. The event has the field `amount_in`, which is defined as the sum of `margin_amount` and `debt_amount`.

The value has an unclear interpretation when the margin token is not equal to the debt token, as balances of two different tokens are summed together.

Code corrected:

The event was removed.

6.20 Floating Pragma

Informational | Version 1 | Code Corrected

CS-UNM-018

Unstoppable uses a floating pragma vyper ^0.3.10. Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Locking the pragma helps to ensure this.

Code corrected:

Unstoppable has fixed the pragma to vyper version 0.3.10.

6.21 SwapRouter Admin Cannot Be Changed

Informational | Version 1 | Code Corrected

CS-UNM-021

The SwapRouter admin role is set once and cannot be transferred to another address. The other contracts contain functionality to change the admin.

Code corrected:

The `suggest_admin` and `accept_admin` functions have been added to the `SwapRouter` contract.

6.22 Vault Uses Incorrect ERC20 Function Interface

Informational | Version 1 | Code Corrected

CS-UNM-025

The Vault defines the incorrect interface for the ERC20 token. The approve function should return true on success.

Note that changing this may break compatability with tokens that incorrectly implement the ERC20 standard, such as USDT on Ethereum mainnet.

Code corrected:

The `approve` function has been removed from the interface defined in Vault.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Arbitrum Sequencer Could Affect Block.Timestamp

Informational Version 1 Acknowledged

CS-UNM-014

The protocol makes use of `block.timestamp`. On Arbitrum, a malicious sequencer is able to change the block timestamps information to 24 hours earlier then the actual time or 1 hour in the future. See also:

https://docs.arbitrum.io/for-devs/concepts/differences-between-arbitrum-ethereum/block-numbers-and-time#block-timestamps-arbitrum-vs-ethereum

If the timestamp value is set to a previous time, the Vault can accept stale Chainlink prices, and MarginDex can execute limit orders that have already expired.

Acknowledged:

Unstoppable is aware of this behavior but accepts the risk due to lack of alternative solutions.

7.2 Arbitrum-Specific Code

Informational Version 1 Acknowledged

CS-UNM-015

Unstoppable is targeting an initial launch to Arbitrum, with plans to extend to additional networks. However, certain parts of the code work exclusively on Arbitrum:

1. The *withdrawTo* function, exclusive to Arbitrum's extended WETH, isn't compatible with other WETH implementations (e.g., Optimism, Mainnet).
2. The addresses for WETH, ARBITRUM_SEQUENCER_UPTIME_FEED, and UNISWAP_ROUTER are hard-coded to Arbitrum's deployment.

Acknowledged:

Unstoppable responded:

```
Part of the deployment checklist.
```

7.3 Avoiding Liquidation Penalty in Self-Liquidation

Informational | Version 2 | Acknowledged

CS-UNM-033

In Version 2, a user's position can be liquidated by anyone if it becomes undercollateralized. The user is then required to pay a liquidation penalty based on the remaining position margin after liquidation. However, if no margin tokens are left, the liquidation penalty becomes zero:

```
# full liquidation, penalty from margin
if position.margin_token != position.debt_token:
    penalty = self._quote(position.debt_token, position.margin_token, penalty)
penalty = min(penalty, position.margin_amount)
```

If a trade returns more debt tokens than required to cover the debt, the excess tokens are credited to the user before any penalty is applied. This means that in a self-liquidation scenario, the user can avoid the liquidation penalty by moving the proceeds from the liquidation into the debt token. Note that the trade must still fulfill the fairness criteria and the check caps the debt change to the minimum position debt amount:

```
amount_in[0] = min(position_debt_amount, convert(_debt_change, uint256))
```

Acknowledged:

Unstoppable responded:

```
Very unlikely that a user is willing and capable to outperform MEV liquidators,
but he doesn't simply close his position before it becomes liquidatable and a
penalty is even in question. Doesn't justify the added complexity to cover this case in code.
```

7.4 Blacklisted Tokens Can Be Credited

Informational | Version 2

CS-UNM-034

Users are prevented from funding their accounts with assets that are not whitelisted in `fund_account` and `fund_account_eth`. But, similar to Blacklisted Tokens Can Be Swapped Into, a user can fund their account through other means: If a user has a debt position in a token that was removed from the whitelist, they can close their position by choosing a route that generates more debt tokens than required to cover their debt. The excess tokens are then credited to the user's margin and can be used to top up existing positions by calling the `change_position` function with *realized_pnl* set to a negative value.

7.5 Cannot Mint or Redeem Shares When Price Is 0

Informational | Version 1 | Acknowledged

CS-UNM-016

The price of an LP share can fall to zero when bad debt is generated. When liquidity is provided or removed, the contract divides by the share price and the transaction reverts.

Acknowledged:

Unstoppable answered:

```
Bad debt would have to be repaid first.
```

7.6 Liquidity Provider Can Withdraw Small Amount of Lent Out Assets

Informational | Version 1 | Acknowledged

CS-UNM-019

According to code comments, Liquidity Providers should not withdraw liquidity that is currently lent out.

The code enforces this by comparing the proposed amount with total liquidity - debt.

However, the contract does not update the debt before making this comparison, so LPs can withdraw a small amount of outstanding interest.

Acknowledged:

Unstoppable responded:

```
Interest accrued is not the same as lent out liquidity.
An LP can withdraw before ``update_debt`` is called, but in exchange
“donates” his interest to the remaining LPs in this case.
```

7.7 No Address 0 Checks

Informational | Version 1 | Code Partially Corrected

CS-UNM-020

The following setter functions do not do a sanity check that the address passed is not zero:

1. `Vault.set_is_whitelisted_dex`: `_dex`
2. `Vault.set_configuration`: `swap_router`
3. `MarginDex.set_vault`: `_vault`

Code partially corrected:

The function `Vault.set_configuration` no longer sets the swap router. The function `MarginDex.set_vault` has been removed from the codebase. The function `Vault.set_is_whitelisted_dex` still does not check for address 0.

Unstoppable responded:

```
Due to code size limitations we had to remove low value checks.
```

7.8 No Grace Period for Sequencer Uptime Feed

Informational | Version 1 | Acknowledged

CS-UNM-036

The Vault retrieves the status code of the Arbitrum Sequencer from a Chainlink oracle. When the sequencer is down, the Vault does not accept the prices provided by Chainlink and reverts.

The Chainlink documentation recommends to wait an additional *grace period* after the sequencer is back up to give users time to improve their leverage.

https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code

Waiting for the grace period increases the risk of price jumps, but could reduce the risk of mass liquidations.

Acknowledged:

Unstoppable responded:

```
The recommendation makes sense for overcollateralized use-cases, in the undercollateralized
environment we are working in, we want to react as quickly as possible instead of waiting
for users who may or may not react manually.
```

7.9 SwapRouter's Exact Output Swap Does Not Handle Non-Standard Tokens

Informational | Version 2 | Acknowledged

CS-UNM-035

In `SwapRouter.swap_exact_out()`, the contract grants approval to the Uniswap Router to spend the maximum amount of tokens. However, the router only withdraws the necessary tokens for the swap, leaving some approval remaining.

This can be problematic with certain non-standard ERC20 tokens (e.g., USDT on Ethereum) that require the approval to be reset to 0 before a new approval can be set. As a result, the router could become locked after the first swap not consuming the full approval. While resetting the approval to 0 post-swap could resolve this, it would render the router contract incompatible with other ERC20 tokens like BNB (on Ethereum) that explicitly prohibit setting the approval to 0.

To accommodate these non-standard tokens, a *safe_approve* wrapper function could be used (see: Transfer in SwapRouter does not handle non-standard tokens).

For more context, refer to the following link: https://github.com/d-xo/weird-erc20?tab=readme-ov-file#approval-race-protections

Note that we are currently not aware of any such tokens on Arbitrum, although they may exist.

Acknowledged:

Unstoppable responded:

```
MarginDEX functionality will be limited for the foreseeable future to a handful
of the main tokens, no plans to add or support exotic/non-standard tokens.
```

7.10 Transfer in SwapRouter Does Not Handle Non-Standard Tokens

Informational **Version 1** **Acknowledged**

CS-UNM-022

The SwapRouter uses the *transfer* function to transfer tokens.

It is considered best practice to use *safe_transfer_from* for transfers, to check the return value and handle non-standard ERC20 tokens, such as those that do not return a boolean value on success (such as USDT and BNB on Ethereum mainnet).

https://github.com/d-xo/weird-erc20?tab=readme-ov-file#missing-return-values

---

Acknowledged:

Unstoppable is aware of this behavior and notes that they have no plans to use non-standard tokens.

7.11 Utilization Rate Is Overestimated

Informational **Version 1** **Acknowledged**

CS-UNM-024

The system accrues debt every second due to interest. However, the interest is paid to Liquidity Provider only when traders close their position or get liquidated. As a result, the contract underestimates the actual available Liquidity and overestimates the utilization ratio (debt / liquidity).

This leads to a higher interest rate than expected.

Acknowledged:

Client is aware of this behavior, but has decided to keep the code unchanged.

7.12 Withdrawing Liquidity Can Intentionally Increase Utilization

Informational **Version 1** **Acknowledged**

CS-UNM-026

Liquidity providers can withdraw their liquidity from the protocol to increase the utilization rate. A larger utilization ratio increases the interest rate charged to borrowers. Rates will stay high until borrowers close positions or additional LPs deposit, both of which will likely take some time. It is crucial to ensure that the maximum interest returned by the interest rate model is not too large.

In the worst case, the interest accrued in a short period (e.g. one block) could be sufficient to liquidate all borrowers.

Acknowledged:

Unstoppable is aware of this behavior.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Behavior in Case of Sequencer Downtime

Note Version 1

Blocks on Arbitrum are ordered by the sequencer. In case this sequencer goes down, blocks will only be created again once it comes back online, or by using the *force-inclusion* mechanism, which can include transactions on L2 through L1, after waiting for the minimum delay of 24 hours.

While the sequencer is down, the Oracle used by the Vault reverts. As a result, it will be impossible to open, close, or modify any position (as these actions call the oracle for a leverage check). In particular, it will also be impossible to liquidate positions. Delaying liquidations may lead to bad debt for the system.

In the special case where the sequencer goes down and never comes back online, users will be able to withdraw from their margin balances as well as the LP using the force-inclusion mechanism. However, it will be impossible to close positions. This means any margin used in a position, as well as any LP funds that are used in active positions will be irrecoverable until the sequencer comes back online. For LPs, this means that there will be a race to withdraw first. Any LPs that withdraw while there are still unutilized funds available will receive their money back, while those that withdraw later when the remaining funds are fully utilized will be unable to withdraw.

Overall, the system is likely to experience bad debt in case the sequencer goes down, especially if it goes down for extended periods.

8.2 Delegates Can Profit From Bad Trades

Note Version 1

According to the trust model, the delegate of a user in MarginDex should be able to trade on their behalf, but they should not be able to withdraw from their balance.

However, a delegate can indirectly steal from the user's balance by making trades with bad slippage parameters and then sandwiching the user, see also Sandwiching order execution.

As a result, the trader should only delegate to addresses they fully trust.

8.3 Interest Is Only Paid When the Positions Are Closed

Note Version 1

Traders accumulate interest continuously, but Liquidity Providers only get paid when positions are closed. If a Liquidity Provider adds funds right before a position closure, they earn interest for the position's entire duration. But if they withdraw right before a closure, they get no interest, even if they were deposited for most of the position's duration.

8.4 Limit Orders Can Become Executable Below Market Price

Note Version 1

Limit orders contain a `min_amount_out` set by the user. This ensures that the order executes at a price that is no worse than the one set by the user. Usually, this is set to a price that is better than the current price.

A limit order that has hit its minimum execution price is only executable if the user has a sufficient margin balance in the Vault. This may not be the case, for example, if the user had another limit order trigger that uses the same margin token. As a result, the limit order will stay open with a price that is now below market price. If the user receives sufficient margin balance again later, (e.g. by closing a position), then the order will be executable again, unless it has passed its `expiry`. The user can get sandwiched and receive only their `min_amount_out`, even though the current market price may be significantly higher.

Users can protect themselves from this in multiple ways:

1. Ensure there is always enough margin to execute all open limit orders.
2. Set the `expiry` of limit orders to a short time
3. Cancel limit orders that have hit their execution price, but cannot be executed due to insufficient margin.

Updated behavior:

In Version 2, a "fairness check" was added to all trades, meaning that in the worst case a stale limit order can now execute at the configured maximum slippage compared to the oracle price.

8.5 Non-Standard ERC-20 Tokens

Note Version 1

We assume that the contract only handles standard ERC-20 Tokens.

We have identified the following issues with non-standard ERC-20 Tokens:

1. The internal accounting of the Vault does not handle rebasing or fee-on-transfer Tokens
2. The price of tokens without `decimals` function cannot be calculated
3. Pausable tokens can stop the Vault from closing positions

8.6 Oracle Manipulation on FIFO L2s

Note Version 1

Unstoppable relies on price oracles for the "fairness check" on swaps, which is critical to the system's security.

Price oracles must be robust and manipulation-resistant. It must be expensive to manipulate the markets which are used as price sources.

Usually, the factor that makes price manipulation expensive is arbitrage. If a manipulator pushes the price of an asset too high or too low, arbitrageurs will see this and make a profit by moving the price back to the "true price". Any profit made by arbitrageurs will be a loss to the manipulator.

This "arbitrage assumption" breaks down in two cases:

1. All markets for the token are manipulated simultaneously, so it is difficult to determine the "true price". There is no other market to arbitrage against.
2. Arbitrageurs are not able to see the manipulated price quickly enough, so they cannot take advantage of it.

Attacks that target condition 2. are known as "Multi-block MEV" attacks. The idea is that a manipulator could control the order of transactions in a block, which allows the following:

- In block `n`, the manipulator sends a transaction (through a private mempool like Flashbots) that manipulates the price of an asset.
- In block `n+1`, the manipulator ensures that the first transaction in the block is one where they revert the price back to the original value.

As a result, arbitrageurs will have no chance of reacting to the manipulation, as it will already be over by the time they can get a transaction included in the block. However, if there is an Oracle that reads the price at the beginning of block `n+1`, it will see the manipulated price.

This attack is well-known on Ethereum, but is generally deemed expensive to execute, as it requires being ,or having an agreement with, the ETH staker that is chosen to propose block `n+1`. If the attack should be repeated multiple times, it requires being chosen as block proposer multiple times within a short time frame, which requires a significant amount of ETH staked.

However, on L2s, block production works differently. Instead of a different proposer being chosen in each block, there is typically a single sequencer that decides on a block ordering policy. One commonly used policy used by chains such as Arbitrum, Optimism and Base is "FIFO" (First In, First Out), where transactions are included in the order they were received.

In FIFO ordering, the order of transactions is determined by time, not by the price a user is willing to pay. This can be taken advantage of to fulfill condition 2. above, without needing to be a block producer.

The FIFO attack looks as follows:

1. The manipulator experiments to figure out their latency to the sequencer (and ideally minimizes it).
2. The manipulator sends a manipulation transaction at a time such that it will arrive at the sequencer towards the end of the period in which it is building block `n`.
3. The manipulator sends a second transaction so that it reaches the sequencer at the beginning of the period in which it is building block `n+1`.

Arbitrageurs are only able to see the manipulated price once block `n` is published by the sequencer. By that time, the manipulator has already sent the second transaction that reverts the price back to the original value. As time is the only relevant factor, it is impossible for a transaction that is created later to be included in the block first (unless the arbitrageur has significantly lower latency to the sequencer).

The only cost to the attacker is the trading fees paid. As the attack cannot use a flashloan, they must also have sufficient capital available to manipulate the price by the percentage they aim for. The attack can be repeated as many times as the attacker wants, although repeated attacks could be speculatively frontrun by arbitrageurs if they detect a pattern. Repeated attacks can be used to circumvent outlier-detection mechanisms and TWAPs.

A policy that modifies transaction ordering to be based on a payment in addition to timing would make the attack significantly more expensive. For example, "Arbitrum time-boost" has been proposed, but not yet implemented. See Time Boost Medium post.

Note that Multi-block MEV attacks have historically been considered mostly in the context of TWAP manipulation. However, if there is an off-chain oracle, such as ChainLink, that uses an on-chain market as a primary price source, the attack also applies there. In fact, the effect will be much larger, as off-chain

oracles typically do not use a time-weighted average. Instead, they read the spot price at a single point in time. As a result, executing the attack once could lead to a heavily manipulated price. Some off-chain oracles may implement outlier-detection to mitigate this, but this is often not clearly documented, if it exists at all. If outlier-detection exists, the attack could be executed multiple times.

In summary, Multi-block MEV attacks are likely much more realistic to execute on FIFO L2s (such as Arbitrum) than on Ethereum, as they are possible without needing to be a block producer. They can affect on-chain TWAPs as well as any off-chain oracles that use L2 on-chain markets as a primary price source. This must be considered when deciding which assets have an oracle that is robust enough to use.

It should be carefully considered before any asset is added to the system for which the price oracle is based primarily on L2 markets.

8.7 Repay Bad Debt Applies to All LPs

Note **Version 1**

The Vault has a `repay_bad_debt` function, that can be used to repay bad debt that was incurred. One use-case of this could be an insurance fund.

Note that when bad debt is repaid, it will be repaid to all LPs of that token that have shares affected by bad debt (usually just the safety module). In particular, it will also apply to LPs that deposited after the bad debt is incurred. Instead of being compensated for their earlier loss, the payment will be a profit to these new LPs.

As a result, the `repay_bad_debt` should be used in a way that is not predictable, as otherwise, it will be profitable to deposit to the LP before the repayment happens.

This is not a problem in case the bad debt is so large that the protocol goes into "defensive mode", which disallows deposits to the LP. The admin could also intentionally activate defensive mode.

8.8 Sandwiching Order Execution

Note **Version 1**

Swaps on Unstoppable Margin Dex, such as opening, closing, or liquidating positions, are always routed through Uniswap. The swaps always have a `min_amount_out` configured to define a maximum allowed amount of slippage.

In some cases, the swap transaction can be executed by anyone. This is the case for limit, take-profit and stop-loss orders, as well as liquidations. In these cases, it is trivial for the executor to "sandwich" the swap with 2 additional swaps on Uniswap that happen in the same transaction, manipulating the execution price. This will cause the swap to execute such that exactly the `min_amount_out` is returned, but no more. The difference to the unmanipulated price will be a profit to the executor. This can be executed using a flash loan, so it has no capital requirements.

For transactions that are only executable by certain addresses (such as a user closing their own position) it is much harder to sandwich, as Arbitrum One does not use a public mempool and has FIFO ordering. However, if it would be possible to tell using some offchain metric that a user is about to execute a trade, they could still be frontrun. On a chain that has a different transaction ordering methodology, it may be possible to reliably frontrun and sandwich these types of transactions too.

It should also be noted that the FIFO ordering means that an arbitrage bot that is optimized for latency should be expected to execute orders before Unstoppable's own off-chain "Liquidation engine" does, as it is likely less optimized.

In summary, for transactions executable by anyone, it should be expected that they clear at the `min_amount_out` price with no surplus.

Updated behavior:

In Version 2 orders are no longer always routed through Uniswap. Instead, they are routed through a caller-provided `P2PSwapper` contract, which can have arbitrary functionality. This means that now a liquidator can ensure that the swap is executed at the worst price that still passes the fairness check, even without manipulating the Uniswap market price. This means the costs of "sandwiching" are now lower. However, Version 2 also introduces partial liquidations, which allows setting a tighter slippage bound for liquidations.

The conclusion that transactions executable by anyone should be expected to clear at the worst acceptable price with no surplus still holds.

A MarginDex implementation that includes an auction mechanism could be added in the future to improve this.

8.9 Slippage Limit Considerations

Note Version 1

The `liquidate_slippage` is a critical parameter that can be set per token pair by the Vault admin. It is used to set the `min_amount_out` of swaps such as liquidations to an amount corresponding to the most recent oracle price minus `liquidate_slippage`.

We illustrate this using the following example:

- The current Oracle price of ETH/USDC is 1000
- The `liquidate_slippage` is set to 5%
- A position of 1 ETH needs to be liquidated
- This will result in a swap with `min_amount_out` of 950 USDC

The `liquidate_slippage` must not be set too large nor too small.

If it is too large, the `min_amount_out` may be less than the amount needed to close the position without creating bad debt. Even if it does not create bad debt, it can lead to bad trade execution for the trader. Note that liquidations should always be expected to clear at `min_amount_out` with no surplus. See also Sandwiching order execution.

If the `liquidate_slippage` is set too small, the swap will revert. This will make it impossible to liquidate the position until the oracle price updates or liquidity on Uniswap improves. Delayed liquidations could lead to bad debt for the system. Slippage being too small should mostly be a problem if a position is very large compared to the liquidity on Arbitrum Uniswap, see also Large Liquidations Can Fail.

To avoid bad debt, `liquidate_slippage` should be set to at most `overcollateralization - oracle_max_move`, where `overcollateralization` can be calculated as `1/(max_leverage+1)`, since liquidations will be triggered once a position hits a leverage ratio of `max_leverage+1`. The `oracle_max_move` is the maximum percentage that the oracle price is expected to move within a single oracle update. For example, if the `max_leverage` on ETH/USDC was 9x (10% overcollateralization) and the oracle price is expected to move at most 5% in one update, then a slippage limit up to 5% would ensure there cannot be any debt caused by a liquidation, as long as the `oracle_max_move` assumption holds.

Note that the above calculation only ensures that the `liquidate_slippage` is not too large, it may still result in a limit that is too small, which could lead to delays in liquidations. Also note that `oracle_max_move` may be difficult to predict in the case of a black swan event such as a flash crash, chain congestion, or the Arbitrum sequencer going down.

Updated behavior:

In Version 2, partial liquidations were introduced. This allows setting tighter slippage limits than behavior, as now a liquidation can be split into multiple smaller trades with lower price impact.

8.10 Unintuitive Max_Leverage Definition

Note Version 1

The `max_leverage` per token pair can be set by the Vault admin. Note that the `max_leverage` is defined as follows:

- Once the leverage of a position reaches `max_leverage + 1`, it is liquidatable.

For example, if `max_leverage` is `10`, a position with `10.99` leverage will not be liquidatable. It will be liquidatable once it reaches leverage `11`.

The admin setting the leverage parameter, as well as anyone analyzing the risk of the system, must take this unintuitive definition into account.

8.11 User Pays Liquidation Penalty on Positive Slippage

Note Version 2

During a liquidation, a penalty is charged as a percentage of the debt tokens paid back. However, the number of debt tokens paid back is not capped by the user's debt and can be slightly larger (when `actual_debt_in > position_debt_amount`).

The user will end up paying the liquidation penalty on any unexpected excess that they receive.