Dolomite Margin Audit Report

Prepared by Cyfrin

Version 3.0

Lead Auditors

0kage

Giovanni Di Siena

Hans

Carlos

Assisting Auditors

Alex Roan

August 26, 2023

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document.

A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

4.1 Overview

Dolomite Margin is a composable margin trading and lending protocol, forked from dYdX Solo, running on Arbitrum One L2. The contracts are not upgradeable, and the protocol enforces base-layer rules for ownership, risk limitations, interest rate accrual, collateralization, and settlement.

4.2 Key Concepts

- *Markets*: Each supported ERC20 token has a dedicated market with a price oracle, interest setter contract, and risk parameters.
- *Collateral-Only Mode*: Certain assets are allowed as collateral but cannot be borrowed against.
- *Virtual Liquidity*: Assets are aggregated into virtual balances for trading, borrowing, and pooling within Dolomite Margin. No on-chain token transfers occur.
- *Capital Efficiency*: Liquidity is recycled to generate multiple sources of yield through internal trading.
- *Supply Caps*: Dolomite Margin administrators can set supply limits to manage debt exposure and borrowing capacity.
- *Minimum Collateralization*: All positions must maintain a minimum collateralization level to avoid liquidation.
- *Isolation Mode*: Allows specific borrowing and collateralization rules and native interactions with underlying tokens.
- *Dynamic Interest Rates*: Rates are algorithmically determined based on the borrowing-to-supply ratio.

4.3 Actions

Actions drive state changes in Dolomite Margin, including deposit, withdraw, transfer, buy, sell, call, liquidate, and vaporise. Internal balances are updated without on-chain token transfers.

4.4 Roles

- *Governance*: Manages the `DolomiteMargin` contract and performs protocol-level actions.
- *Users*: Execute actions on their accounts or designated operator accounts.
- *Operators*: Self-operators, local operators approved by users, or global operators approved by the protocol admin.

5 Executive Summary

Over the course of 25 days, the Cyfrin team conducted an audit on the Dolomite Margin smart contracts provided by Dolomite. In this period, a total of 15 issues were found.

Cyfrin conducted an audit of the Dolomite project based on the code present in the repository commit hash e10f143. Contracts present in the following files/directories were included in the audit scope:

```
- contracts/protocol
- contracts/external/proxies/BorrowPositionProxyV1/V2
- contracts/external/proxies/DepositWithdrawalProxy
- contracts/external/proxies/GenericTraderProxyV1
- contracts/external/proxies/LiquidatorProxyV4WithGenericTrader
- contracts/external/proxies/MarginPositionRegistry
```

The codebase consists of three layers: the core layer, the implementation layer, and the proxy layer. The core layer, derived from the dY/dX solo margin protocol, proved robust during the audit. The implementation layer contains libraries that call functions from the core layer, categorized by underlying actions such as admin actions, trading, deposit, withdrawal, borrowing, liquidation, etc. The main contract, `DolomiteMargin`, utilizes the state defined in `Storage` and the implementation libraries to allow users to define generic operations.

During the audit, 15 findings were discovered, five of which were graded `Medium` severity. One issue relates to the lack of systemic risk controls to hedge potential insolvency risks, particularly during black-swan liquidity freeze events. Given Dolomite's vision of becoming a base layer for trading a wide range of tokens, we recommend additional risk controls to better handle contagion events.

Dolomite Margin relies heavily on Chainlink Oracles, but it was found that deprecated functions were being used to fetch the latest asset prices. Additionally, the current handling of L2 sequencer downtime poses a risk to borrowers if significant price movements occur during the downtime. An accounting issue was identified in the `TradeImpl::buy` function, which failed to account for scenarios where the protocol receives more tokens than expected. Dolomite has successfully addressed both these issues.

The audit further highlighted a potential insolvency risk stemming from the protocol admin's unintentional withdrawal of excess tokens. Additionally, there was a distinct concern regarding the protocol admin's capacity to withdraw unsupported tokens with double entry points, which could lead to significant imbalances in Dolomite's balances Dolomite has recognized both concerns and intends to implement safeguards to address these scenarios.

While Dolomite has a documentation website, the technical documentation, especially for the proxy contracts, was found to be incomplete at times. Extensive communication with the Dolomite team was necessary to address higher-level questions and concerns. Furthermore, some contracts, like `GenericTraderProxyV1`, relied on logic abstracted to interfaces whose implementations were not part of the scope, hindering business logic verification. It is recommended that the team improve the technical documentation regarding Isolation Mode Tokens, wrapping/unwrapping actions, and expiry contracts.

Overall, addressing the identified issues, improving documentation, and refining risk controls will enhance the security and reliability of the Dolomite Margin protocol.

Summary

Project Name	Dolomite Margin
Repository	v2
Commit	e10f14320ece...
Audit Timeline	June 12th - July 14th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	5
Low Risk	6
Informational	4
Gas Optimizations	0
Total Issues	15

Summary of Findings

[M-1] Chainlink price and L2 sequencer uptime feeds are not used with recommended validations and guardrails	Resolved
[M-2] Admin can drain market of double-entrypoint ERC-20 using `AdminImpl::ownerWithdrawUnsupportedTokens`	Acknowledged
[M-3] Inaccurate accounting in `TradeImpl::buy` could lead to loss of user funds	Resolved
[M-4] `AdminImpl::ownerWithdrawExcessTokens` does not check the solvency of given market before attempting to withdraw excess tokens	Acknowledged
[M-5] Inadequate systemic risk-controls to support cross-asset collateralization across a wide range of assets	Acknowledged
[L-1] Incorrect logic in `Bits::unsetBit` when applied to a zero bit	Resolved
[L-2] OpenZeppelin v2.5.1 is not supported for non-critical security patches	Acknowledged
[L-3] It may be possible to exploit external call in `CallImpl::call`	Acknowledged
[L-4] Violation of the Checks Effects Interactions Pattern in `LiquidateOrVaporize::liquidate` and `LiquidateOrVaporize::vaporize` could result in read-only reentrancy vulnerabilities	Resolved

[L-5] Lack of max value validation in `AdminImpl::ownerSetAccountMaxNumberOfMarketsWithBalances`	Resolved
[L-6] Inadequate checks and high trust assumptions while setting global operators	Acknowledged
[I-1] Repeated logic can be reused with a shared internal function	Resolved
[I-2] `BorrowPositionProxyV2` allows authorized global operators to unilaterally modify positions and transfer debt between accounts	Acknowledged
[I-3] Inconsistencies in project README.md	Resolved
[I-4] Unverifiable contracts resulting from logic abstraction in externally scoped implementations	Acknowledged

6 Findings

6.1 Medium Risk

6.1.1 Chainlink price and L2 sequencer uptime feeds are not used with recommended validations and guardrails

Description: `ChainlinkPriceOracleV1` is an implementation of IPriceOracle which is used in `Storage::fetchPrice`. The protocol currently validates price cannot be zero, but there exist no checks for staleness and round incompleteness which could result in use of an incorrect non-zero price. `ChainlinkPriceOracleV1` currently uses the deprecated `IChainlinkAggregator::latestAnswer` function instead of the recommended `IChainlinkAggregator::latestRoundData` function in conjunction with these additional validations.

L2 sequencer downtime validation is handled by calls to a contract conforming to the IOracleSentinel interface in both `OperationImpl::_verifyFinalState` and `LiquidateOrVaporizeImpl::liquidate`. The contract on which `getFlag` is called simply returns the sequencer uptime status and nothing else.

When an L2 sequencer comes back online after a period of downtime and oracles update their prices, all price movements that occurred during downtime are applied at once. If these movements are significant, borrowers rush to save their positions, while liquidators rush to liquidate borrowers. Since liquidations are in the future intended to be handled by Chainlink Automation, without some grace period where liquidations are disallowed, borrowers are likely to suffer mass liquidations. This is unfair to borrowers, as they could not act on their positions even if they wanted to due to the L2 downtime.

Impact:

1. Lack of staleness and round-incompleteness validation could result in the use of an incorrect non-zero price.
2. Lack of a sequencer downtime grace period could mean that borrow positions become immediately liquidatable once the sequencer is back up and running if there is a large price deviation in the intermediate time period.

Recommended Mitigation: The Dolomite Margin protocol should correctly validate values returned by Chainlink data feeds and give borrowers a grace period to deposit additional collateral prior to allowing liquidations to resume after a period of L2 sequencer downtime.

Dolomite: Fixed as of commit 6a8ae06.

Cyfrin: Acknowledged.

6.1.2 Admin can drain market of double-entrypoint ERC-20 using `AdminImpl::ownerWithdrawUnsupportedTokens`

Description: `AdminImpl::ownerWithdrawUnsupportedTokens` is intended to allow the owner to withdraw any unsupported ERC-20 token which might have ended up at the Dolomite Margin address. If a double-entrypoint ERC-20 token is listed as a market on Dolomite, it is possible for the admin to drain the entire token balance.

Such tokens are problematic because the legacy token delegates its logic to the new token, meaning that two separate addresses are used to interact with the same token. Previous examples include TUSD which resulted in vulnerability when integrated into Compound. This highlights the importance of carefully selecting the collateral token, especially as this type of vulnerability is not easily detectable. In addition, it is not unrealistic to expect that an upgradeable collateral token could become a double-entrypoint token in the future, e.g. USDT, so this must also be considered.

By passing the legacy token address of a double-entrypoint token as argument to `AdminImpl::ownerWithdrawUnsupportedTokens`, the admin can drain the entire token balance. The legacy token will not have a valid market id as it has not been added to Dolomite, so `AdminImpl::_requireNoMarket` will pass.

```
function ownerWithdrawUnsupportedTokens(
    Storage.State storage state,
    address token,
    address recipient
)
    public
    returns (uint256)
{
    _requireNoMarket(state, token);

    uint256 balance = IERC20Detailed(token).balanceOf(address(this));
    token.transfer(recipient, balance);

    emit LogWithdrawUnsupportedTokens(token, balance);

    return balance;
}
```

However, function calls on the legacy token will be forwarded to the new version, so the balance returned will be that of the token in the protocol, which will be transferrable to the admin.

Impact: This finding would have a critical impact, leaving the protocol in an insolvent state at the expense of its users; however, the likelihood is low due to external assumptions, and so we evaluate the severity as MEDIUM.

Recommended Mitigation: Loop through all supported Dolomite Margin markets and validate the collateral token balances before withdrawing unsupported tokens are equal to the token balances after.

Dolomite: Since Dolomite's core protocol is built to support listing hundreds (or thousands) or assets, we don't want to implement the proposed fix. First, we don't intend to list any tokens with this strange behavior. Second, the proposed fix could cause us to run out of gas for a given block since iterating through the balances for each token will get really costly as the number of assets listed grows.

We may provide a hot fix for it in the future through an ownership adapter that adds this functionality.

Cyfrin: Acknowledged.

6.1.3 Inaccurate accounting in `TradeImpl::buy` could lead to loss of user funds

Description: Within Dolomite Margin, `TradeImpl::buy` is used to perform a buy trade and takes `Actions.BuyArgs memory args` as one of its parameters. Given these arguments are supplied by the caller, they are free to set any arbitrary value for `args.exchangeWrapper`.

This `args.exchangeWrapper` parameter is used to calculate the amount of `takerWei` the contract should receive, and to calculate how much should actually be transferred to the Dolomite contract.

Given it is not guaranteed that these two values will be the same, the issue arises when there is a difference between the amount that should be received versus the actual amount received. This is partially handled in `TradeImpl::buy`, which ensures the received amount is greater than or equal to the expected amount, acting as a slippage check. However, the internal accounting is updated based on the value expected to be received, instead of the value actually received.

Impact: Incorrect accounting could result in loss of user funds. Given that it is expected, but not guaranteed, that `takerWei == tokensReceived`, we evaluate the severity to MEDIUM.

Recommended Mitigation: Modify these lines to guarantee a correct update to protocol accounting.

```
//  TradeImpl::buy
    Require.that(
        tokensReceived.value >= makerWei.value,
        FILE,
        "Buy amount less than promised",
        tokensReceived.value
    );

-   state.setPar(
-       args.account,
-       args.makerMarket,
-       makerPar
-   );
+   state.setParFromDeltaWei(
+       args.account,
+       args.makerMarket,
+       makerIndex,
+       tokensReceived
+   );
```

Dolomite: Fixed as of commit 6a8ae06.

Cyfrin: Resolved.

6.1.4 `AdminImpl::ownerWithdrawExcessTokens` does not check the solvency of given market before attempting to withdraw excess tokens

Description: `AdminImpl::ownerWithdrawExcessTokens` allows the protocol admin to withdraw excess tokens in Dolomite Margin for a specific market. Excess tokens are calculated using the following formula:

```
Excess tokens = Token Balance (L) + Total Borrowed (B) - Total Supplied (S)
```

Here, `L` represents the real liquidity, which is the actual token balance in Dolomite Margin. `B` and `S` are virtual Dolomite balances. Over time, excess tokens increase as the protocol earns fees after passing the borrowers' interest fee to suppliers. The extent of these fees depends on the total outstanding borrowing in the market and the earnings rate.

However, in certain scenarios, the admin's withdrawal of `numExcessTokens` can lead to temporary insolvency in the protocol. This occurs when the token balance remaining after withdrawal is lower than the maximum withdrawable value for that market after adjusting for collateralization. This situation is especially likely in less liquid markets or in markets with high concentration risk, where a single entity has provided a significant portion of liquidity.

Impact: In this situation, withdrawal actions might cause a denial-of-service (DoS) due to the protocol's inadequate balance. While the consequences could be significant, the chance of this happening is minimal since fees are typically much lower than pool balances. As a result, we assess the severity level as MEDIUM.

Proof of Concept: *Assumptions:*

- The interest rate on USDC is 10% per year
- Earnings rate = 80% (20% of borrowing interest is the protocol fee)

Consider a simplified scenario below:

Time	Action	Alice	Bob	Pete	USDC Balance	ETH Balance	Excess USDC
T =0	Alice deposits 2 ETH, Bob 5000 USDC	2	5000	-	5000	2	0
T=0	Alice borrows 2000 USDC	2 , -2000	5000	-	5000	2	0

Time	Action	Alice	Bob	Pete	USDC Balance	ETH Balance	Excess USDC
T=1yr	200 USDC interest accrued	2, -2200	5160	-	5000	2	40
T=1yr	Pete deposits 1000 USDC	2, -2200	5160	1000	6000	2	40
T=1yr	Bob withdraws 5160 USDC	2, -2200	0	1000	840	2	40
T=1yr	Protocol withdraws 40 USDC	2, -2200	0	1000	800	2	0

At this stage, Pete cannot withdraw his deposit even though there is no loan against his account. This is because the protocol is temporarily insolvent until another liquidity provider deposits fresh liquidity.

Recommended Mitigation: To address this issue, it is recommended to introduce solvency checks for each market in the `AdminImpl::ownerWithdrawExcessTokens` function before completing withdrawals. The protocol should ensure that withdrawing excess tokens does not result in temporary insolvency.

Dolomite: A way to mitigate this in the future (without any code changes to the core protocol) is to have the admin withdraw their tokens and atomically redeposit them. This would enable the admin to "compound" their earnings, keeping liquidity in the protocol, while still enabling the admin to "zero out" their excess tokens.

Cyfrin: Acknowledged.

6.1.5 Inadequate systemic risk-controls to support cross-asset collateralization across a wide range of assets

Description: Current Dolomite Margin risk controls can be classified into two categories:

- Account-level risk controls
- Systemic risk controls

Dolomite incorporates a robust risk monitoring architecture that comprehensively verifies the final state's validity at the conclusion of each operation. An operation, which encompasses a collection of transactions, undergoes thorough system checks to ensure the integrity and accuracy of the final state. This calculation happens in `OperationImpl::_verifyFinalState` where both account-level and system-level risk is measured.

While the existing systemic risk controls address various aspects, such as borrowing limits, supply limits, collateral-only mode, and oracle sentinel, they fail to consider the systemic risk introduced by cross-asset collateralisation.

The creation of virtual liquidity without sufficient token backing can expose the protocol to risks associated with liquidity squeezes, freezes, and fluctuating asset correlations. This is similar to fractional banking in traditional finance, i.e., if a bank lends more than 10x its deposits, the bank exposes itself to insolvency risks due to tight liquidity conditions.

In the case of Dolomite Margin, the ratio of virtual liquidity to actual token balance can be considered as leverage - the higher the leverage, the greater the insolvency risk. If this virtual liquidity is utilised as collateral for additional borrowing, it can further amplify the leverage.

Impact: Higher protocol leverage can directly increase the risk of insolvency during periods of tight liquidity. Since such risks are attributed to extremely rare black-swan type of events, we evaluate the severity to MEDIUM.

Proof of Concept: Consider a USDT de-peg scenario where we see the following events unfold:

1. Virtual USDT holders will scramble to convert to real USDT liquidity, attempting to offload it anywhere possible.
2. Speculators, seeing an opportunity, will initiate cross-collateral borrow positions in USDT. High potential profits in short periods can make them overlook even steep interest rates at higher utilisation levels.
3. These virtual USDT tokens can then be swiftly exchanged for other stable tokens within Dolomite's internal pools.

4. Liquidators might hesitate to liquidate positions with USDT as collateral, anticipating potential high slippage in one-sided markets.

All the above factors may trigger a cascading effect, leading Dolomite to accumulate substantial USDT bad debt. While borrow/supply limits exist, introducing additional safeguards to curb unbacked liquidity could better equip Dolomite to manage potential contagion scenario.

Recommended Mitigation: To enhance the system's robustness, consider adding a systemic risk measure to the `OperationImpl::_verifyFinalState` function that caps the leverage per market. Consider implementing a cap on leverage for less liquid markets restricting the ability of users to open cross asset borrowing positions without having enough real liquidity.

Dolomite: We think this is better managed through the use of a Global Operator. Each situation has so much nuance to it. So the ability to force close, or add mechanisms that can modify positions will be really helpful in tackling nearly any situation

Cyfrin: Acknowledged. While Global Operators certainly bolster Dolomite's agility in making rapid account adjustments, their responsiveness and efficacy in widespread crises remain ambiguous. Past events, such as the Terra Luna incident, have shown that speculators often exploit de-peg situations, making aggressive bets that potentially saddle the protocol with significant bad debt. We continue to recommend additional risk controls that limit the creation of unbacked virtual liquidity to mitigate any existential threats during such rare contagion scenarios.

6.2 Low Risk

6.2.1 Incorrect logic in `Bits::unsetBit` when applied to a zero bit

Description: `Bits::unsetBit` might not work as expected for cases when the bit to be unset is not 1.

It is currently implemented as:

```
function unsetBit(
    uint bitmap,
    uint bit
) internal pure returns (uint) {
    return bitmap - (ONE << bit);
}
```

This implementation will only correctly unset a bit if that bit is already set to 1. If the bit were set to 0, due to the way in which binary subtraction works, this operation would instead set it to 1 and modify other bits as well (which is unlikely to be the intended behavior).

A more appropriate way to unset a bit would be to use a bitwise `AND` operation with the complement of the bit mask. The corrected implementation would look like this:

```
function unsetBit(
    uint bitmap,
    uint bit
) internal pure returns (uint) {
    return bitmap & ~(ONE << bit);
}
```

In this version, `(ONE << bit)` creates a mask where only the bit at position bit is set to 1. The `~` operator then inverts this mask, setting the bit at position bit to 0 and all other bits to 1. Finally, the bitwise `AND` operation leaves all bits in the bitmap unchanged, except for the bit at position bit, which is unset (set to 0).

Impact: It appears this function is only ever called with set bits due to `Bits::getLeastSignificantBit` being called beforehand in both instances; otherwise, this finding would have a much higher impact, but we evaluate the severity to LOW.

Recommended Mitigation: Use the modified version of the function above.

Dolomite: Fixed as of commit 6a8ae06.

Cyfrin: Acknowledged.

6.2.2 OpenZeppelin v2.5.1 is not supported for non-critical security patches

The Dolomite Margin smart contracts currently use OpenZeppelin contracts v2.5.1, while the latest release is v4.9.2. Per the OpenZeppelin security policy, only critical severity bug fixes will be backported to past major releases. In light of this, it is possible that future bug reports or changes to the protocol may introduce vulnerabilities and so it is recommended to use a more up-to-date version.

Dolomite: In order to maintain compatibility with the rest of Dolomite's smart contracts which use Solidity v5, we use the latest major version of OpenZeppelin v2. If we ever decide to upgrade the smart contracts, we'll be sure to bump the OpenZeppelin version as well.

Cyfrin: Acknowledged.

6.2.3 It may be possible to exploit external call in `CallImpl::call`

Description: `CallImpl::call` contains the following logic:

```
state.requireIsOperator(args.account, msg.sender);
ICallee(args.callee).callFunction(
    msg.sender,
    args.account,
    args.data
);
```

Selector clashing and/or fallback function for a given callee can be triggered here with arbitrary data in the context of `DolomiteMargin`, assuming the sender is an operator for the given account. `DolomiteMargin` libraries are deployed as standalone contracts, so it appears `OperationImpl` will be `msg.sender` in the context of the call.

Fortunately, this cannot be used to exploit the infinite WETH approval in `DepositWithdrawalProxy::initializeETHMarket` as there are no clashing selectors and the WETH fallback function simply attempts to deposit `msg.value`.

Impact: This finding could have critical severity under certain circumstances but depends on a number of external assumptions, which significantly reduce the likelihood, and so we evaluate the severity to LOW.

Recommended Mitigation: Consider whitelisting the trusted contracts that can be used as `args.callee`.

Dolomite: We'd prefer to keep the implementation as permissionless as possible. If there's another possible check we can put in place, we'd gladly explore it.

Cyfrin: Acknowledged.

6.2.4 Violation of the Checks Effects Interactions Pattern in `LiquidateOrVaporize::liquidate` and `LiquidateOrVaporize::vaporize` could result in read-only reentrancy vulnerabilities

Description: When liquidating or vaporizing an undercollateralized account, there are a number of instances where SafeLiquidationCallback::callLiquidateCallbackIfNecessary is called, temporarily handing off execution to the liquid account owner. While there are precautions in place to mitigate against gas-griefing and return data bombing attacks, this logic fails to prevent vulnerabilities in third-party protocols which may rely on state from Dolomite Margin. Updates to state balances are performed after these calls, so whilst it is acknowledged in the interface and cannot be exploited directly given the global protocol-level reentrancy guard, this may expose integrating protocols to a read-only reentrancy attack vector.

Impact: There is no immediate impact on Dolomite Margin, but this finding could impact other ecosystem protocols which may be affected by intermediate protocol state, so we evaluate the severity as LOW.

Recommended Mitigation: Update state balances prior to making unsafe external calls.

Dolomite: We're adding the callback mechanism to all Actions that modify the user's virtual balance but don't materialize an ERC20 transfer event. To standardise it, they were added before the Action's events are logged. Those actions are `Transfer` , `Trade`, `Liquidate` , and `Vaporize`. We also added a variable called `callbackGasLimit` that lets the admin set the amount of gas to allocate to the callback functions. This allows for more control over deployments where gas may not be measured the same way.

Fixes added as of commit 6a8ae06.

Cyfrin: Acknowledged. In the specified commit, we noted changes unrelated to this audit concern, particularly changes related to callbacks and the introduction of callbackGasLimit replacing the prior hardcoded gas. Our review focused solely on the consistency of the CEI pattern in the liquidate and vaporize functions, without delving into the wider implications of these modifications.

6.2.5 Lack of max value validation in `AdminImpl::ownerSetAccountMaxNumberOfMarketsWithBalances`

Description: Currently, there is no protection to stop the admin from setting a value greater than an upper bound when calling `AdminImpl::ownerSetAccountMaxNumberOfMarketsWithBalances`, only the condition that it should be at least 2:

```
Require.that(
    accountMaxNumberOfMarketsWithBalances >= 2,
    FILE,
    "Acct MaxNumberOfMarkets too low"
);
```

This could lead to a denial-of-service scenario that is intended to be prevented, so there should be an additional maximum validation.

Impact: There is low likelihood due to reliance on admin error and so we evaluate the severity as LOW.

Recommended Mitigation: Add additional validation for the upper bound when the owner sets the maximum allowed number of markets with balances for a given account.

Dolomite: Fixed as of commit 6a8ae06. Added an upper bound of 64 markets.

Cyfrin: Acknowledged.

6.2.6 Inadequate checks and high trust assumptions while setting global operators

Description: Dolomite introduces the concept of `Global Operators`, which are accounts authorised to perform actions on behalf of other users. The purpose of these accounts is to enhance user experience by minimising the need for constant transaction approvals. However, it is important to recognise the potential risks associated with malicious global operators who possess privileged access. Two instances in the code highlight these risks:

Malicious global operators, because of their privileged access, can potentially cause a lot of damage. Two such instances observed in the code are:

1.Global operators can create new borrow positions on behalf of users using the `BorrowPositionProxyV2::openBorrowPositionWithDifferentAccounts` function

2. Global operators implementing `IAutoTrader` can act as market makers for users, engaging in trading activities with their taker counterparts.

Upon discussions with the protocol team, it was clarified that only specific contracts are assigned the Global Operator role, with no intention to designate externally owned accounts (EOAs) as Global Operators. However, considering the significant control wielded by global operators, the current checks and controls for assigning this role are inadequate.

Impact: A malicious global operator has the capability to manipulate funds by engaging in unauthorized trading or borrowing against user accounts.

Recommended Mitigation: To strengthen the security of the system, the following additional checks are advised when assigning a global operator:

- Explicitly check that global operator is not a Externally Owned Account (EOA).
- Implement a time-lock mechanism when registering a new global operator.
- Establish a whitelisting mechanism, allowing only specific contracts within the whitelisted universe to be assigned as global operators.

Dolomite: Dolomite maximizes being generic on the core level and we've tried to put in place as many safeguards as possible for input validation.

The protocol is currently governed by a delayed multi sig and eventually a time-locked DAO (implementation not complete as of now). This enables the protocol to time-gate everything and leave the details to match the needs of the protocol depending on who owns it.

We're not going to add the EOA check because there are valid use cases in the future for setting a create2 address as a global operator, which would not be possible anymore with the proposed check.

If for any reason control of the protocol is hijacked, the hijacker can create a malicious operator contract which would be just as bad as an EOA, anyway.

Cyfrin: Acknowledged.

6.3 Informational

6.3.1 Repeated logic can be reused with a shared internal function

`BorrowPositionProxyV1::openBorrowPosition` and `BorrowPositionProxyV1::transferBetweenAccounts` both contain the following repeated code:

```
AccountActionLib.transfer(
    DOLOMITE_MARGIN,
    /* _fromAccountOwner = */ msg.sender, // solium-disable-line
    _fromAccountNumber,
    /* _toAccountOwner = */ msg.sender, // solium-disable-line
    _toAccountNumber,
    _marketId,
    Types.AssetAmount({
        sign: false,
        denomination: Types.AssetDenomination.Wei,
        ref: Types.AssetReference.Delta,
        value: _amountWei
    }),
    _balanceCheckFlag
);
```

Consider moving this to a shared internal function to reduce bytecode size.

Dolomite: Fixed as of commit 6a8ae06.

Cyfrin: Acknowledged.

6.3.2 `BorrowPositionProxyV2` allows authorized global operators to unilaterally modify positions and transfer debt between accounts

`BorrowPositionProxyV2` contains functions for opening/closing/transferring/repaying borrow positions with different accounts when called by an authorized sender. Whilst this is the intended behavior of the protocol with its global operators and this specific proxy, it is worth noting that this does allow authorized operators to transfer debt from one account to another unilaterally.

Dolomite: We use this functionality to cross boundaries of accounts owned by the same user. For example, Isolation Mode vaults are essentially smart contract wallets owned by an EOA. We use the BorrowPositionProxyV2 so the user can transfer funds from their account to their vault.

Cyfrin: Acknowledged.

6.3.3 Inconsistencies in project README.md

The project README states that `numberOfMarketsWithBorrow` field has been added to `Account.Storage`, but it should instead be `numberOfMarketsWithDebt`.

Additionally, it states that a require statement has been added to `OperationImpl` that forces liquidations to come from a global operator, but this should be `LiquidateOrVaporizeImpl`.

Similarly, the require statement that forces expirations to come from a global operator has been added to `TradeImpl::trade` and not `OperationImpl`.

Consider adding links to relevant lines of code to make navigating these changes easier.

Dolomite: Fixed as of commit 6a8ae06.

Cyfrin: Acknowledged.

6.3.4 Unverifiable contracts resulting from logic abstraction in externally scoped implementations

Description: Dolomite is a versatile base layer that facilitates various actions across multiple tokens. The current audit scope encompasses contracts such as `GenericTraderProxyV1.sol` and `LiquidatorProxyV4WithGenericTrader.sol`. These contracts rely on interfaces like `IIsolationModeUnwrapperTrader::createActionsForUnwrapping` and `IIsolationModeWrapperTrader::createActionsForWrapping` within key functions such as `GenericTraderProxyBase::_appendTraderActions`. Similarly, the interface function `IIsolationModeToken.isTokenConverterTrusted` is utilised in `GenericTraderProxyBase::_validateIsolationModeStatusForTraderParam`.

During the audit, the absence of wrapper/unwrapper trader implementations hindered our ability to verify the precise logic behind creating wrapping and unwrapping actions. The incorrect sequencing or execution of these actions may introduce vulnerabilities that impact the business logic of the mentioned contracts. Additionally, the lack of isolation mode token contracts within the audit's scope prevented us from examining the rationale behind token converter trust. Consequently, we are unable to verify critical functionalities of certain proxy contracts.

Impact: Interacting with these interfaces may lead to unforeseen issues that we are currently unable to comprehend.

Recommended Mitigation: To address this, it is crucial to provide the necessary implementations for the missing contracts and functions.

Dolomite: Acknowledged.

Cyfrin: Acknowledged.

7 Appendix

7.1 4naly3er Static Analysis

Cleaned output from the 4naly3er tool.

7.1.1 Gas Optimizations

[GAS-1] Using bools for storage incurs overhead: Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

Instances (3):

```
File: protocol/lib/Storage.sol

183:         mapping (address => mapping (address => bool)) operators;

186:         mapping (address => bool) globalOperators;

189:         mapping (address => bool) specialAutoTraders;
```

[GAS-2] Cache array length outside of loop: If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Instances (14):

```
File: external/proxies/BorrowPositionProxyV1.sol

91:         for (uint256 i = 0; i < _collateralMarketIds.length; i++) {
```

```
File: external/proxies/BorrowPositionProxyV2.sol

102:         for (uint256 i = 0; i < _collateralMarketIds.length; i++) {
```

```
File: external/proxies/GenericTraderProxyV1.sol

263:         for (uint256 i = 0; i < _param.transferAmounts.length; i++) {
```

```
File: protocol/Permission.sol

62:         for (uint256 i = 0; i < args.length; i++) {
```

File: protocol/impl/GettersImpl.sol

```
654:         for (uint256 i = 0; i < markets.length; i++) {
751:         for (uint256 i = 0; i < markets.length; i++) {
```

File: protocol/impl/OperationImpl.sol

```
123:         for (uint256 a = 0; a < accounts.length; a++) {
124:             for (uint256 b = a + 1; b < accounts.length; b++) {
154:         for (uint256 i = 0; i < actions.length; i++) {
198:         for (uint256 a = 0; a < accounts.length; a++) {
201:             for (uint256 i = 0; i < marketIdsWithBalance.length; i++) {
236:         for (uint256 i = 0; i < actions.length; i++) {
385:         for (uint256 a = 0; a < accounts.length; a++) {
```

File: protocol/lib/Storage.sol

```
916:         for (uint i = 0; i < cache.marketBitmaps.length; i++) {
```

[GAS-3] Don't initialize variables with default value:

Instances (22):

File: external/proxies/BorrowPositionProxyV1.sol

```
91:         for (uint256 i = 0; i < _collateralMarketIds.length; i++) {
```

File: external/proxies/BorrowPositionProxyV2.sol

```
102:         for (uint256 i = 0; i < _collateralMarketIds.length; i++) {
```

File: external/proxies/GenericTraderProxyV1.sol

```
263:         for (uint256 i = 0; i < _param.transferAmounts.length; i++) {
314:         for (uint256 i = 0; i < _transferActionsLength; i++) {
```

File: protocol/Permission.sol

```
62:         for (uint256 i = 0; i < args.length; i++) {
```

File: protocol/impl/GettersImpl.sol

```
654:         for (uint256 i = 0; i < markets.length; i++) {

751:         for (uint256 i = 0; i < markets.length; i++) {
```

File: protocol/impl/OperationImpl.sol

```
123:         for (uint256 a = 0; a < accounts.length; a++) {

154:         for (uint256 i = 0; i < actions.length; i++) {

198:         for (uint256 a = 0; a < accounts.length; a++) {

201:             for (uint256 i = 0; i < marketIdsWithBalance.length; i++) {

208:         for (uint i = 0; i < cache.getNumMarkets(); i++) {

236:         for (uint256 i = 0; i < actions.length; i++) {

320:         for (uint256 i = 0; i < numMarkets; i++) {

385:         for (uint256 a = 0; a < accounts.length; a++) {
```

File: protocol/lib/Require.sol

```
349:         for (uint256 i = 0; i < 20; i++) {

382:         for (uint256 i = 0; i < 32; i++) {
```

File: protocol/lib/Storage.sol

```
484:         for (uint256 i = 0; i < numMarkets; i++) {

753:         bool hasNegative = false;

755:         for (uint256 i = 0; i < numMarkets; i++) {

911:         uint counter = 0;

916:         for (uint i = 0; i < cache.marketBitmaps.length; i++) {
```

[GAS-4] `++i` **costs less gas than** `i++`**, especially when it's used in** `for`**-loops (**`--i`/`i--` **too):** *Saves 5 gas per loop*

Instances (27):

File: external/proxies/BorrowPositionProxyV1.sol

```
91:         for (uint256 i = 0; i < _collateralMarketIds.length; i++) {
```

File: external/proxies/BorrowPositionProxyV2.sol

```
102:         for (uint256 i = 0; i < _collateralMarketIds.length; i++) {
```

File: external/proxies/GenericTraderProxyV1.sol

```
263:         for (uint256 i = 0; i < _param.transferAmounts.length; i++) {
314:         for (uint256 i = 0; i < _transferActionsLength; i++) {
315:             _actions[_cache.actionsCursor++] = AccountActionLib.encodeTransferAction(
338:         _actions[_cache.actionsCursor++] = AccountActionLib.encodeExpirationAction(
```

File: external/proxies/LiquidatorProxyV4WithGenericTrader.sol

```
221:             _actions[_genericCache.actionsCursor++] = AccountActionLib.encodeExpiryLiquidateAction(
233:             _actions[_genericCache.actionsCursor++] = AccountActionLib.encodeLiquidateAction(
```

File: protocol/Permission.sol

```
62:         for (uint256 i = 0; i < args.length; i++) {
```

File: protocol/impl/GettersImpl.sol

```
654:         for (uint256 i = 0; i < markets.length; i++) {
751:         for (uint256 i = 0; i < markets.length; i++) {
```

```
File: protocol/impl/OperationImpl.sol

123:         for (uint256 a = 0; a < accounts.length; a++) {

124:             for (uint256 b = a + 1; b < accounts.length; b++) {

154:         for (uint256 i = 0; i < actions.length; i++) {

198:         for (uint256 a = 0; a < accounts.length; a++) {

201:             for (uint256 i = 0; i < marketIdsWithBalance.length; i++) {

208:         for (uint i = 0; i < cache.getNumMarkets(); i++) {

236:         for (uint256 i = 0; i < actions.length; i++) {

320:         for (uint256 i = 0; i < numMarkets; i++) {

385:         for (uint256 a = 0; a < accounts.length; a++) {
```

```
File: protocol/lib/Require.sol

308:             length++;

349:         for (uint256 i = 0; i < 20; i++) {

382:         for (uint256 i = 0; i < 32; i++) {
```

```
File: protocol/lib/Storage.sol

484:         for (uint256 i = 0; i < numMarkets; i++) {

755:         for (uint256 i = 0; i < numMarkets; i++) {

916:         for (uint i = 0; i < cache.marketBitmaps.length; i++) {

924:                 cache.markets[counter++] = Cache.MarketInfo({
```

[GAS-5] Using `private` rather than `public` for constants, saves gas: If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Instances (1):

```
File: protocol/interfaces/IPriceOracle.sol

35:     uint256 public constant ONE_DOLLAR = 10 ** 36;
```

[GAS-6] Use shift Right/Left instead of division/multiplication if possible:

Instances (1):

```
File: protocol/lib/Cache.sol

170:         uint mid = beginInclusive + len / 2;
```

[GAS-7] Use != 0 instead of > 0 for unsigned integer comparison:

Instances (24):

```
File: external/proxies/GenericTraderProxyV1.sol

245:             _param.transferAmounts.length > 0,

265:                 _param.transferAmounts[i].amountWei > 0,
```

```
File: external/proxies/LiquidatorProxyV4WithGenericTrader.sol

143:         constants.expiryProxy = _expiry > 0 ? EXPIRY: IExpiry(address(0)); // don't read EXPIRY;
↪  it's not needed

220:         if (_constants.expiry > 0) {
```

```
File: protocol/lib/Bits.sol

67:         return bit > 0;

83:         if (x & uint128(-1) > 0) {

89:         if (x & uint64(-1) > 0) {

95:         if (x & uint32(-1) > 0) {

101:         if (x & uint16(-1) > 0) {

107:         if (x & uint8(-1) > 0) {

113:         if (x & 0xf > 0) {

119:         if (x & 0x3 > 0) {

126:         if (x & 0x1 > 0) {
```

```
File: protocol/lib/Cache.sol

109:             cache.markets.length > 0,
```

```
File: protocol/lib/Require.sol

271:         for (uint256 i = 32; i > 0; ) {

317:         for (uint256 i = length; i > 0; ) {
```

```
File: protocol/lib/Storage.sol

784:         if (marginRatioOverride.value > 0 && liquidationSpreadOverride.value > 0) {

784:         if (marginRatioOverride.value > 0 && liquidationSpreadOverride.value > 0) {
```

```
File: protocol/lib/Token.sol

91:         if (returnData.length > 0) {
```

```
File: protocol/lib/Types.sol

158:         return !a.sign && a.value > 0;

168:         return a.sign && a.value > 0;

188:         return a.value > 0 && !a.sign;

297:         return !a.sign && a.value > 0;

307:         return a.sign && a.value > 0;
```

7.1.2 Non Critical Issues

[NC-1] Constants should be defined rather than using magic numbers:

Instances (2):

```
File: protocol/lib/Require.sol

342:         bytes memory result = new bytes(42);

375:         bytes memory result = new bytes(66);
```

7.1.3 Low Issues

[L-1] Empty Function Body - Consider commenting why:

Instances (3):

File: external/proxies/BorrowPositionProxyV1.sol

```
52:     {}
```

File: external/proxies/BorrowPositionProxyV2.sol

```
55:     {}
```

File: external/proxies/DepositWithdrawalProxy.sol

```
79:     {}
```