Quantstamp

Levva Protocol

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	Margin Trading and Derivatives
Timeline	2023-08-28 through 2023-09-15
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Marginly Documentation
Source Code	• https://github.com/eq-lab/Marginly #d82fefd
Auditors	• Ruben Koch Auditing Engineer • Shih-Hung Wang Auditing Engineer • Cameron Biniamow Auditing Engineer

Documentation quality	Medium
Test quality	Medium
Total Findings	24 Fixed: 16 Acknowledged: 8
High severity findings	5 Fixed: 5
Medium severity findings	3 Fixed: 3
Low severity findings	7 Fixed: 4 Acknowledged: 3
Undetermined severity findings	1 Acknowledged: 1
Informational findings	8 Fixed: 4 Acknowledged: 4

Summary of Findings

Note: Since the audit, the client rebranded "Marginly" to "Levva", hence the report refers to the protocol as "Marginly".

Marginly is a margin trading and derivatives platform that allows users to provide liquidity to the protocol while gaining interest, deposit collateral, open long/short positions, and liquidate unhealthy positions to ensure overall system health.

The audit team found five high-severity and three medium-severity issues that indicate that the system is not functioning as intended. For example, user and protocol funds are at risk due to inadequate slippage protection that makes swaps susceptible to sandwich attacks. Most of the issues found revolve around the complexity and novel design for calculating user/protocol collateral and debt. This is exacerbated due to a lack of in-depth technical documentation regarding these mechanisms. It is highly recommended to improve documentation to explain the functionality of all protocol mechanisms clearly.

Regarding the project quality, the code is well-written, and test coverage is moderate. The auditors followed a best-effort approach to identify potential combinations of inputs and outputs that could lead to unexpected behavior. Our team frequently interacted with the Marginly team to clarify code and expected behavior. Their active engagement in answering our questions was crucial and greatly assisted in completing the audit.

We strongly recommend that the client address and consider all the findings in this report.

Update (First Fix Review): Most findings and best practices have been addressed for the fix review, which has been provided under commit hash 422e7398d4d82ae63c973f49952589c1f2a81fc3. However, the provided fix for issue `MAR-2` does not properly handle the case where unhealthy positions are present during an emergency shutdown. Therefore, if this were to be the case, users could withdraw more tokens than they are entitled to.

Update (Second Fix Review): The Marginly team has now fixed, mitigated, or sufficiently acknowledged all issues within the report, which has been provided under commit hash 2e69e9a9bdfa05199e03cb75a493730b98a0f048.

ID	DESCRIPTION	SEVERITY	STATUS
MAR-1	**Vulnerability to Sandwich Attacks Due to Ineffective Slippage Checks**	• High	Fixed

ID	DESCRIPTION	SEVERITY	STATUS
MAR-2	Unfair Distribution of Left-over Collateral in Case of Emergency Withdrawals	High	Fixed
MAR-3	Max-Heap Property Can Break when Calling `MaxBinaryHeapLib.remove()`	High	Fixed
MAR-4	Max-Heap Property Can Break when Calling `receivePosition()`	High	Fixed
MAR-5	Lack of User Restrictions in Shutdown Mode	High	Fixed
MAR-6	Denial-of-Service via Massive Collateral Deposits	Medium	Fixed
MAR-7	System Leverage Can Briefly Exceed Max Leverage	Medium	Fixed
MAR-8	Lack of Slippage Control in Emergency Shutdown	Medium	Fixed
MAR-9	Privileged Roles and Ownership	Low	Acknowledged
MAR-10	Automatic Liquidation Is Limited to One Position per Block	Low	Acknowledged
MAR-11	Risks Associated with Utilizing Uniswap V3 Twap Oracles	Low	Acknowledged
MAR-12	The Use of `SafeERC20.safeApprove()`	Low	Fixed
MAR-13	Critical Role Transfer Not Following Two-Step Pattern	Low	Fixed
MAR-14	Missing Input Validation	Low	Fixed
MAR-15	Ownership Can Be Renounced	Low	Fixed
MAR-16	Unrelated Protocol Interactions Can Revert Due to Liquidation Slippage Being Exceeded	Informational	Acknowledged
MAR-17	`receivePosition()` Mechanic Can Absorb All Liquidation Profits From Pool	Informational	Acknowledged
MAR-18	Fee-on-Transfer or Rebasing Tokens Not Supported	Informational	Acknowledged
MAR-19	`MarginlyParams.quoteLimit` Can Be Exceeded	Informational	Acknowledged
MAR-20	Unlocked Pragma	Informational	Fixed
MAR-21	Application Monitoring Can Be Improved by Emitting More Events	Informational	Fixed
MAR-22	Owner Can Set Arbitrary Fees	Informational	Fixed
MAR-23	Clone-and-Own	Informational	Fixed
MAR-24	Inaccurate Accounting of System Coefficients Due to Precision Loss	Undetermined	Acknowledged

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

> **Disclaimer**
>
> Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
    1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
    1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

Files Included

- `packages/contracts/contracts/*`
- `packages/router/contracts/*`

Files Excluded

- `packages/contracts/contracts/test/*`
- `packages/router/contracts/test/*`

Findings

MAR-1

Vulnerability to Sandwich Attacks Due to Ineffective Slippage Checks

High Fixed

> **Update**
>
> Marked as "Fixed" by the client.
> Addressed in: `9d73c12d39f97a81737c5bbb6d72808431b3a6ef` .
> The client provided the following explanation:
>
> ```
> We've added limitprice argument to the smart contract for open/close. For liquidations we use TWAP with
> short time period (parameter of the smart-contract)
> ```

File(s) affected: `MarginlyPool.sol`

Description: Marginly pools execute swaps through the router when a user creates a long or short position, closes their position, or their position is liquidated. The pool calculates a minimum output or maximum input token amount based on the spot price of a Uniswap V3 pool. The router then enforces that the actual received or input amount falls within the limits.

However, slippage checks based on spot prices of Uniswap V3 pools may be ineffective, as the price impact of a user's trade can be relatively small when the pool has enough liquidity. In other words, the larger the pool's liquidity, the less significant the price impact tends to be. This ineffectiveness is exacerbated by Uniswap V3's centralized liquidity design, where specific tick ranges can have particularly deep liquidity, further reducing the potential price impact of the user's trade.

As the slippage checks are ineffective in protecting users from significant price changes, the user may suffer from sandwich attacks. In such attacks, the attacker manipulates token balances in a Uniswap V3 pool, thereby influencing the outcomes of swaps. As a result, users may receive fewer output tokens or may be required to transfer more input tokens to the pool than expected. The possible consequences in each scenario are:

- Open position: Users might have a position with a higher leverage level than intended.
- Close position: Users may receive fewer tokens than expected.
- Liquidation: The swapped amount may fall short of covering the debt, causing the lenders to absorb the losses instead of resulting in profit.

Exploit Scenario:
The following are the detailed attack steps when a user opens a position:

1. Suppose that the spot price in the Uniswap V3 ETH/USDC pool is 1 ETH = 1000 USDC, and the Marginly pool references this pool for the ETH/USDC spot price.
2. A user wants to open a short position in the Marginly pool, borrowing 1 ETH from the pool while receiving 1000 USDC in return (assuming no additional swap or protocol fees for simplicity).
3. The user sends their transaction to a public mempool.
4. An attacker front-runs the user's transaction and manipulates the spot price in favor of USDC. The manipulation results in the user receiving only 500 USDC after the swap.

The attacker then back-runs the user's transaction to restore the pool state to its previous state.

Consequently, the user receives less USDC than expected, leading to an unintended increase in their position's leverage.

Recommendation: Consider avoiding using the spot price of Uniswap V3 pools to implement slippage checks. Instead, consider allowing users to specify minimum output or maximum input token amounts as parameters for operations involving swaps. The user would then have the option to adjust the swap limits according to the market conditions.

MAR-2
Unfair Distribution of Left-over Collateral in Case of Emergency Withdrawals

High | Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `bdefe8a85ef98988b4865f0e39b78103488ac2be` and `5fc9d87060ecd18e99aaa79b5dff3367b47816ec`.
The client provided the following explanation:

> `We fixed distribution to account for net position (collateral – debt) and not the collateral`

File(s) affected: `MarginlyPool.sol`

Description: In case the collateral of the long or short position is ever exceeded by their respective debt, the pool can be shut down from its regular use and put into a so-called emergency mode by a privileged role.

A short emergency, for example, happens when the base debt exceeds the quote collateral. As a result, all quote tokens of the protocol are sold off and, together with the remaining base assets of the protocol, will be withdrawable by all long positions and all base asset lenders. The withdrawable amount for the pool's leftover funds of each position is essentially proportional to the position's share of base collateral in relation to the pool's total base collateral.

As leveraged long positions have a very high `discountedBaseAmount`, as their leveraged base ownership is recorded in that field, their proportional share is up to 20x (i.e. max leverage) higher than for base lenders or long positions without leverage.

It is our opinion that the emergency fund distribution should be based on the collateral deposited to the protocol and not based on possibly leveraged amounts.

The fairness is also not the only concern here. The emergency mode activation e.g. a short emergency could be front-run by a max leverage long position that sells off all remaining quote collateral (without the `MarginlyParams.quoteLimit` being exceeded for the base token) that could manage to get a great share of the whole protocol's funds that way.

It should also be mentioned that due to all quote collateral being sold off and made accessible to the base lenders and longers, even the base lenders or the unleveraged longs could profit from such emergencies (specifically if `emergencyCoefficient > 1`).

Recommendation: The share of emergency withdrawals should be proportional only to the position's unleveraged deposited collateral.

MAR-3
Max-Heap Property Can Break when Calling `MaxBinaryHeapLib.remove()`

High | Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `2a6c007b3db66f31d9b892e23c137103b558c2fe` .
The client provided the following explanation:

```
Fixed as per issue recommendation
```

File(s) affected: `MaxBinaryHeapLib.sol`

Description: In the `MaxBinaryHeapLib.remove()` function, the heap replaces the to-be-removed node with the last node (i.e., the node with the largest index) and invokes `heapifyDown()` with the inserted node to adjust its position in the heap. However, the inserted node may need to be pushed toward the root by calling `heapifyUp()` instead. In such cases, the inserted node will not be placed in the correct position in the heap, which could lead to a failure in maintaining the maximum node in subsequent operations.

The max heap aims to track the pool's riskiest short or long position. When the `reinit()` function is invoked, it only evaluates whether the position at the top of the heap is unhealthy, and if so, the position will get liquidated. As a result, the failure of the heap could prevent the liquidation of an unhealthy position if a healthy position is at the top of the heap.

Note that liquidators can still trigger the liquidation of an unhealthy position by directly calling `receivePosition()` . However, if the liquidators are absent, the system may be unable to execute the liquidation by itself due to this bug.

Exploit Scenario:
To illustrate the potential issue, consider the following sequence of operations:

1. Insert node A with a key of 1008.
2. Insert node B with a key of 1003.
3. Insert node C with a key of 1006.
4. Insert node D with a key of 1002.
5. Insert node E with a key of 1000.
6. Insert node F with a key of 1005.
7. Remove node D.
8. Update node A to 1001.

Subsequently, we pop out the top element from the heap five times. The nodes are popped from the heap in the following order: C, B, F, A, and E. However, since node F has a key greater than that of node B, node F should have been popped out before node B.

Recommendation: Consider modifying the `remove()` function so that after the last node replaces the old node, it is swapped upwards or downwards in the heap based on its value relative to the old node's value. The adjustment should be similar to the approach implemented in the `update()` function.

MAR-4 Max-Heap Property Can Break when Calling `receivePosition()`

High | Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `441a4ce1b025ebd686ae89ac1e38229b1488ffa0` .
The client provided the following explanation:

```
Fixed as per issue recommendation
```

File(s) affected: `MarginlyPool.sol`

Description: In the `MarginlyPool` contract, the `receivePosition()` function allows users to receive unhealthy positions while choosing the amount of debt to repay (and the amount of collateral to add) to restore position health. This flexibility can result in changes to the position's leverage.

However, since no `updateHeap()` is called at the end of the `receivePosition()` function, the `sortKey` of the user's position remains unchanged, even when the position leverage decreases. As a result, the position is incorrectly positioned in the heap, potentially preventing other unhealthy positions from being liquidated by `reinit()` due to the inconsistent sorting.

Exploit Scenario:
Consider the following scenario:
1. Users A and B open a short position, with A having a higher leverage level. The current heap is [A, B], with A at the top of the heap (index 0).
2. The price of the base asset has increased, making both A and B's positions unhealthy.

3. User C comes and partially receives A's position. However, since the position's `sortKey` is unchanged, it remains in its original position in the heap. Now, the heap becomes [C, B].
4. Although user B's position remains unhealthy, it will not get liquidated as the `reinit()` function only checks the health of the top element of the heap, which is C's position.

Recommendation: Consider invoking `updateHeap()` whenever a position's leverage has changed, which ensures the correct updating of the position's `sortKey`. Similarly, if a position should be removed, ensure the `remove()` function is invoked to maintain heap consistency.

MAR-5 Lack of User Restrictions in Shutdown Mode

High Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `259f7f59ced68bc2305ffa32acf416dbf823a15a`.
The client provided the following explanation:

```
Added proposed checks for the shutdown mode
```

File(s) affected: `MarginlyPool.sol`

Description: In scenarios with system insolvency, i.e., the total collateral value is insufficient to support the entire debt value on the opposite side, the protocol admin will invoke `MarginlyPool.shutdown()`. According to the official documentation:

```
In the shutdown mode, no new positions are allowed to be opened, and the side opposite to the
insolvent side is responsible for restoring the system's solvency at its expense.
```

However, according to the code implementation, users can still open new positions even if the pool is in shutdown mode, as the code does not explicitly prevent the users from doing so.

Recommendation: Consider adding a check that verifies the state variable `mode` is set to `Mode.Regular` at the start of non-emergency-related functions. This check will help ensure that users cannot create or change their positions during the shutdown mode, except for the emergency withdrawal, aligning with the intended protocol behavior as documented.

MAR-6 Denial-of-Service via Massive Collateral Deposits

Medium Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `0e60fb2dd453ef8fdeaca14ac240efa3861eb409` and `bf9febf02f74c776e4eff87f378044c40d1e60bd`.
The client provided the following explanation:

```
Added soft limit as proposed
```

File(s) affected: `MarginlyPool.sol`

Description: If a user wants to keep their leveraged long or short position from being liquidated, they might need to deposit additional collateral to lower the position's leverage. Before the collateral deposit is accepted, it is checked that the pool's total collateral value (denominated in the quote asset price), together with this deposit, does not exceed `MarginlyParams.quoteLimit`. Else, the transaction reverts.

This opens up the possibility of a DoS-attack vector, wherein a malicious actor possessing substantial financial resources strategically deposits both the quote and base assets as a lending position into the protocol with the goal of elevating the protocol's asset balances for both the quote and base assets to reach `MarginlyParams.quoteLimit`.

Unless profitable positions are closed, or liquidations occur, these values will not change, making the active position's leverage unadjustable. As most liquidations absorbed by the pool will profit collateral holders, there is an incentive for malicious actors to perform such an attack.

This could also happen in more natural, non-malignant conditions: With a falling base asset price, many shorts would close, increasing the base balance in the protocol. If such closing positions reach the quote limit, long positions cannot keep their positions healthy.

Recommendation: As the `MarginlyParams.quoteLimit` parameter is a fairly soft limit anyways (see issue "`MarginlyParams.quoteLimit` Can be Exceeded"), we propose not to check for it when shorters are attempting to deposit base assets, and longers attempt to deposit quote assets.

MAR-7 System Leverage Can Briefly Exceed Max Leverage

Medium Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `37f780258396f89f4a471f9318aad7e9cbeb4a8f`.
The client provided the following explanation:

```
Capped max leverage in the accrued interest rate calculation.
```

File(s) affected: `MarginlyPool.sol`

Description: The Marginly Pool can hold multiple to-be-liquidated positions at once. If protocol usage is slow and bots are not participating in liquidations via `receivePosition()`, the total debt of one position's side can, therefore, briefly exceed the collateral of a position by a factor greater than the max leverage.

This is especially note-worthy in situations where the pool would be about to shut down, i.e., where the base debt exceeds the quote collateral. If such a shutdown call would be front-run by a user calling `receivePosition()` on a few liquidatable short positions (i.e., increasing quote collateral), they could perform enough liquidations to stop the short-emergency from being activated. In such a scenario, the system leverage could be almost arbitrarily high (assuming, e.g., base debt is equal to a hundred thousand USD and quote collateral equal to a hundred thousand and one USD would update the system short leverage to a hundred thousand), making the interest rate for shorts for the period possibly vastly higher than anticipated.

The conditions for this would, however, be improbable to happen. Not only would the pool need to be close to an emergency state, but an attacker would need to be perfectly organized to profit from such an attack (i.e., have a massive long position in place to receive the fees and front-run the emergency activation with a perfectly orchestrated series of `receivePosition()` calls).

Recommendation: In `MarginlyPool.accrueInterest()`, consider capping the system leverage used for interest calculations at the max leverage for user trades.

MAR-8 Lack of Slippage Control in Emergency Shutdown

Medium | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `ea24c1f400c7fc5541624fa415f1f6ccfe35b1ca`.
The client provided the following explanation:

```
Added new argument in shutdown mode. Now able to route shutdown liquidation to different liquidity
venues.
```

File(s) affected: `MarginlyPool.sol`

Description: In the event of a short emergency within a Marginly ETH/USDC pool, where the total ETH debt value exceeds the total USDC collateral value, the protocol admin initiates a pool shutdown and sets the mode to `ShortEmergency` by invoking the `setEmergencyMode()` function. During the execution of this function, all USDC collateral minus the USDC debt is swapped into ETH using the Marginly router.

However, the swap is executed with the minimum output token amount set to 0, which makes the swap susceptible to sandwich attacks. In the most severe scenario, the pool may receive nearly 0 ETH in return, with most of the value extracted by the malicious attacker.

Recommendation: Consider spreading the swap across multiple DEXs, determining the swap amount for each DEX based on their liquidity. This will reduce the overall price impact of the swap and the likelihood of the swap being sandwich-attacked.

MAR-9 Privileged Roles and Ownership

Low | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
We will add the governance roadmap to our docs, where we explicitly state that at the current stage all
of the parameter changes are controlled by the owner key.
```

File(s) affected: `MarginlyFactory.sol, MarginlyPool.sol, RouterStorage.sol, AdapterStorage.sol`

Description: Smart contracts often have `owner` variables to designate the person with special privileges to modify the smart contract. If a malicious user compromises or controls the contract owner, sensitive functionality can be exploited. The following are all instances of functionality restricted to the contract owner or a privileged role:

- Only the `owner` address of the `MarginlyFactory` contract can:

    - call `MarginlyFactory.createPool()`, deploying new pools with specified `MarginlyParams` and linked with a specific Uniswap pool.
    - call `MarginlyFactory.changeSwapRouter()`, replacing the existing swap router. A replacement with a malicious swap router could drain all input tokens from the protocol users' requested swaps.
    - call `MarginlyFactory.setOwner()`, transferring the contract ownership to any address, including the zero address.
- Only the factory `owner` can
    - call `MarginlyPool.setParameters()` at all times, which can potentially severely misconfigure the pool. Updated values include the maximum leverage, the TWAP configuration, the basic interest rate, the fee ("close debt fee") and swap fee, the slippage thresholds, the minimum size of a position, and the discussed `quoteLimit`.
    - call `MarginlyPool.shutdown()`, shutting down the pool and setting it to the proper emergency mode when either long's or short's collateral is exceeded by its respective debt.
    - call `MarginlyPool.sweepETH()`, receiving all the protocol's ETH. It should be noted that the internal base asset is not expected to be ETH but WETH, so it does not enable the withdrawal of core protocol pool assets.
- Only the `owner` role of the `MarginlyRouter` contract can:
    - call `RouterStorage.addDexAdapters()`, enabling the addition and overwriting of swap adapters, i.e., intermediary contracts that settle the swap requests. A compromised adapter could forward to-be-swapped funds to arbitrary addresses.
- Only the `owner` role of the specific adapters in use can:
    - call `AdapterStorage.addPools()`, where new pools can be added or existing pools overridden for a certain DEX adapter. A malicious address added as a pool could forward to-be-swapped funds to arbitrary addresses.

Furthermore, the Marginly team is expected to take over slippage-exceeding liquidations in turbulent market conditions. If this would not happen, the pool could become insolvent.

Exploit Scenario:
Particularly, a malicious or compromised `owner` of `FullMarginlyFactory` or `MarginlyFactory` can change the swap router to a malicious contract. Given that the minimum output or maximum input checks are not enforced by the `MarginlyPool` contract (but by the adapter contracts on the router side), a malicious router can execute an unfavorable trade and potentially extract the user's funds for profit.

Further, a malicious swap router contract could take advantage of any remaining allowances that users may have previously granted and transfer tokens directly from the user.

Recommendation: Consider documenting the risks associated with compromised or malicious privileged roles in public-facing documentation. As privileged roles can be the single point of protocol failure, consider using a multi-sig or timelock to mitigate the risk of being compromised or exploited.

MAR-10
Automatic Liquidation Is Limited to One Position per Block

Low | Acknowledged

Update
Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
We've explicitly covered the limitation in the docs:
  https://docs.marginly.com/protocol-mechanics/trading#liquidations
```

File(s) affected: `MarginlyPool.sol`

Description: In the current design of the Marginly pool, the `reinit()` function only checks whether the position at the top of the heap (i.e., the riskiest position) is unhealthy. Even though other positions in the heap are unhealthy, they will not get liquidated since the `reinit()` function is executed only once per block. As a result, the `reinit()` function can liquidate at most one unhealthy position in each block, even if multiple positions within the heap are unhealthy.

Although liquidators can directly trigger the liquidation of any unhealthy position in the system by utilizing `receivePosition()`, this design presents potential challenges in scenarios where liquidators are absent and price changes rapidly. If the unhealthy position fails to be liquidated timely, the system may incur bad debt.

Recommendation: Consider clarifying this risk in public-facing documentation to ensure that users are aware of the system's behavior. Also, ensure the keeper bot operated by the protocol team actively monitors the positions in each pool and responds promptly to liquidate unhealthy positions. This proactive approach can help mitigate the risk of system insolvency.

MAR-11
Risks Associated with Utilizing Uniswap V3 Twap Oracles

Low | Acknowledged

Update
Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
Added a section about TWAP oracles into our docs: https://docs.marginly.com/protocol-mechanics/risk-management#twap-oracle
```

File(s) affected: `MarginlyPool.sol, MarginlyFactory.sol`

Description: Marginly pools rely on Uniswap V3 pools as TWAP oracles to perform calculations related to position health, system leverage, pool balance limits, etc. As a result, the resistance of TWAP oracles to manipulation is critical to the system, as manipulated prices can have significant consequences for both the system and its users. This issue highlights the potential risks associated with utilizing Uniswap V3 TWAP oracles:

1. Uniswap V3 pools with low liquidity are more susceptible to manipulation, as attackers require less capital to influence prices. Since protocol admins can create Marginly pools using Uniswap V3 pools of different fee tiers, they should be cautious when selecting fee tiers associated with pools with less liquidity.
2. The design of concentrated liquidity of Uniswap V3 pools can make TWAP manipulations more cost-efficient as liquidity concentrated within specific price ranges may reduce the costs caused by price slippage.
3. The presence of multi-block MEV introduces risks of TWAP manipulation. As block producers, attackers can leverage control over consecutive blocks, thereby increasing the ease of price manipulation and avoiding the risk of being arbitrated by external actors.
4. An insufficient TWAP period may make the calculated prices more susceptible to manipulation.

Recommendation:

1. Ensure that Uniswap V3 pools used as TWAP oracles contain sufficient liquidity spread across the entire price range, effectively increasing the cost of TWAP manipulation.
2. Consider implementing active monitoring of TWAP prices and respond promptly to any potential price manipulation activities.

MAR-12 The Use of `SafeERC20.safeApprove()`

• Low Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `115098f80cac923bd18daddc2db1d4fbd7887596` .
The client provided the following explanation:

```
Implemented as per issue recommendation
```

Description: The code relies on the function `SafeERC20.safeApprove()` . This function has been deprecated and shall not be used instead of `approve()` since 1) it can still be front-run, 2) it uses additional gas, and 3) it does not work with tokens that are not ERC-20 compliant. For more information, see this issue.

- `MarginlyKeeper.liquidateAndTakeProfit()` uses `IERC20.approve()` and `SafeERC20.safeApprove()` , which is generally not recommended.
- The other files use Uniswap's `TransferHelper.safeApprove()` , which still has the issue of being unable to change a currently active approval. Some places in the code call `TransferHelper.safeApprove(0)` to solve this issue, but some don't.

Recommendation: Consider using OpenZeppelin's `SafeERC20.forceApprove()` , which solves the above-mentioned issues.

MAR-13 Critical Role Transfer Not Following Two-Step Pattern

• Low Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `5224430a9be0bbd98a3a21a0ed482d097d20c446` .
The client provided the following explanation:

```
Implemented as per issue recommendation
```

File(s) affected: `FullMarginlyFactory.sol, MarginlyFactory.sol, AdapterStorage.sol, RouterStorage.sol`

Description: The owner of the factory contract can call `setOwner()` , and the owner of the `AdapterStorage` and `RouterStorage` can call `transferOwnership()` to transfer the ownership to a new address. Suppose an uncontrollable address is accidentally provided as the new owner's address. In that case, the contract will no longer have an active owner, and functions restricted to the contract owner can no longer be executed.

- `FullMarginlyFactory` and `MarginlyFactory` have a custom owner implementation and use `setOwner()` to transfer the ownership, which does not follow the two-step pattern.
- `AdapterStorage` and `RouterStorage` inherit OpenZeppelin's `Ownable` contract.

Recommendation: Consider using OpenZeppelin's `Ownable2Step` contract to adopt a two-step ownership pattern in which the new owner must accept their position before the transfer is complete.

MAR-14 Missing Input Validation

Low | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `ea70e673a5b61aadfff9c3d86f7fc05cfa7f6351` and `2e69e9a9bdfa05199e03cb75a493730b98a0f048`.
The client provided the following explanation:

```
Implemented as per issue recommendation
```

File(s) affected: `FullMarginlyFactory.sol, MarginlyPool.sol`

Related Issue(s): SWC-123

Description: It is important to validate inputs, even if they only come from trusted addresses, to avoid human error. Specifically, in the following functions, arguments of type `address` may be initialized with value `0x0`:

- `FullMarginlyFactory.constructor()`:
  - `uniswapFactory`
  - `swapRouter`
  - `feeHolder`
  - `WETH9`
  - `techPositionOwner`
- `FullMarginlyFactory.setOwner()`
  - `owner`
- `MarginlyPool._initializeMarginlyPool()`
  - `quoteToken`
  - `baseToken`
  - `quoteTokenIsToken0`
  - `uniswapPool`
  - `params`
- `MarginlyPool.setParameters()`
  - `params`

Recommendation: We recommend adding the relevant checks.

MAR-15 Ownership Can Be Renounced

Low | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `1759ca6613f41864bec78d54b88e3bdc67763099`.
The client provided the following explanation:

```
Implemented as per issue recommendation
```

File(s) affected: `MarginlyFactory.sol`

Description: If the owner renounces ownership, all ownable contracts will be left without an owner. Consequently, any function guarded by the `onlyOwner` modifier will no longer be able to be executed. The functions listed under *Privileged Roles and Ownership* will no longer be executable if contract ownership is renounced.

Recommendation: Validate that the input is not the zero address for the function `MarginlyFactory.setOwner()`. Consider using a two-step process for extra security when transferring the contract ownership (e.g., `Ownable2Step` from OpenZeppelin) and override the `renounceOwnership()` function to revert when called.

MAR-16 Unrelated Protocol Interactions Can Revert Due to Liquidation Slippage Being Exceeded

Informational | Acknowledged

Update

Such protocol interactions will still revert, even though a more generous slippage is applied now. Specifically, liquidations may revert if the price quickly changes, and exceeds the slippage rate. The Marginly team sees it as their responsibility to overtake liquidations that would cause the liquidation slippage to be exceeded.

Update

Marked as "Fixed" by the client.
Addressed in: `9d73c12d39f97a81737c5bbb6d72808431b3a6ef`.
The client provided the following explanation:

```
Fixed as per MAR-1, will update docs accordingly
```

File(s) affected: `MarginlyPool.sol`

Description: Liquidations have a slippage threshold based on the `params.mcSlippage` value applied to the `slot0()` price in the respective pool. The liquidation call reverts if the slippage is exceeded. The Marginly team is expected to overtake illiquid positions in such situations. We want to raise awareness that the slippage checks may delay or entirely prevent the admins from liquidating these illiquid positions when slippage is too high, and the swaps fail. Thus, the illiquid positions could become more unhealthy and impact overall system health.

Exploit Scenario:
Assume the price of ETH quickly drops, many long positions become unhealthy, and network activity spikes as users attempt to sell off their ETH.

1. The transactions fail When the positions are liquidated since the price falls faster than the maximum slippage rate.
2. The positions become more unhealthy as ETH prices continue to drop.
3. Repeat steps 1 - 2 as admins attempt to liquidate the unhealthy positions again.

Recommendation: Consider using a sufficiently high slippage rate for illiquid liquidations or remove the slippage check entirely to prevent reverted liquidations.

MAR-17
`receivePosition()` Mechanic Can Absorb All Liquidation Profits From Pool

Informational | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
We stated this explicitly in our docs. https://docs.marginly.com/protocol-
mechanics/trading#liquidations
```

File(s) affected: `MarginlyPool.sol`

Description: With the `receivePosition()` function, any user can take over a liquidatable position from another user, given that they provide at least enough collateral to keep the position healthy. According to the documentation, the function seems to have mainly been designed for illiquid collateral situations, but it is worth noting that it can be called at any time.

In most market conditions, receiving positions will be very lucrative for bots to monitor and execute, as they could immediately realize the 5% liquidation bonus. Therefore, we do not think the pool will receive the 5% liquidation bonus regularly. The bots will not call the `receivePosition()` function for insolvent positions, so only the unprofitable liquidations will be left to the pool or our client's deployed instance of the monitor bot.

While these bots will increase the system's stability with timely liquidations, they will most likely receive most of the liquidation profits.

This has been acknowledged by the client, who stated that ensuring several means of protection from liquidations is crucial and that the primary source of revenue for LPs will most likely come from the borrowing activity and not the liquidations.

Recommendation: Adjust the documentation on docs.marginly.com to reflect that most of the profits from LPs will come from the accumulating fees and that most of the liquidation profits will most likely be received by liquidation bots.

MAR-18
Fee-on-Transfer or Rebasing Tokens Not Supported

Informational | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
Will update docs (overview section) accordingly.
```

File(s) affected: `Adapter Contracts`

Description: The design of all the adapters does not support fee-on-transfer tokens. Furthermore, the pool's accounting would not work well using rebasing tokens as the quote or base asset. Therefore, such tokens shall not be used for the protocol.

Recommendation: We mainly want to raise awareness of this fact. Consider adding appropriate documentation for potential third-party forks.

MAR-19 `MarginlyParams.quoteLimit` Can Be Exceeded

Informational | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> ```
> OK, will add to docs section related to limit parameters.
>   (Limits may be exceeded on close position, liquidations and debt repay).
> ```

File(s) affected: `MarginlyPool.sol`

Description: The `MarginlyParams.quoteLimit` parameter indicates the maximum amount of assets denominated in the quote asset price that the protocol can hold separately for both the quote and base asset. It is checked for before performing asset deposits and opening (or adding to) a long or short position.

We want to point out that this is more of a soft limit. It can be exceeded in natural protocol conditions, e.g., when positions are closed, and in case of direct transfers to the protocol followed by a balance syncing.

Recommendation: We mainly want to raise awareness for this. We did not identify any resulting security risks.

MAR-20 Unlocked Pragma

Informational | Fixed

> **Update**
> Marked as "Fixed" by the client.
> Addressed in: `13118e6e40c6054e357caddf6b842c3430e374b7`.
> The client provided the following explanation:
>
> ```
> Solidity version 0.8.19
> ```

Related Issue(s): SWC-103

Description: Every Solidity file specifies a version number of the format `pragma solidity (^)0.8.*` in the header. The caret (`^`) before the version number implies an unlocked pragma, meaning the compiler will use the specified version *and above*, hence the term "unlocked".

Recommendation: For consistency and to prevent unexpected behavior in the future, we recommend to remove the caret to lock the file onto a specific Solidity version.

MAR-21 Application Monitoring Can Be Improved by Emitting More Events

Informational | Fixed

> **Update**
> Marked as "Fixed" by the client.
> Addressed in: `b7f855d94d901f4a3018013391997aa968fbed57`.
> The client provided the following explanation:
>
> ```
> Fixed as per issue recommendation
> ```

Description: Emitting events is a good practice to validate the proper deployment and initialization of the contracts. Also, any important state transitions can be logged, which is beneficial for monitoring the contract and tracking eventual bugs or hacks. Below, we present a non-exhaustive list of events that could be emitted to improve application management:

- `FullMarginlyFactory.changeSwapRouter()` should emit the new swap router address.
- `MarginlyFactory.changeSwapRouter()` should emit the new swap router address.
- `MarginlyPool.setParameters()` should emit the global pool parameters.
- `RouterStorage.addDexAdapter()` should emit the address of the DEX adapter added.
- `AdapterStorage.addPools()` should emit the pool address added.

Recommendation: Consider emitting the events.

MAR-22 Owner Can Set Arbitrary Fees

Informational | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `ea70e673a5b61aadfff9c3d86f7fc05cfa7f6351`.
The client provided the following explanation:

```
Overlaps with MAR-14
```

File(s) affected: `MarginlyPool.sol`

Description: The owner can set arbitrary fees via the function `MarginlyPool.setParameters()`. Specifically, the owner can set the following fees and rates up to and exceeding 100%.

- `MarginlyPool.params.interestRate` can be set above 100% (1,000,000).
- `MarginlyPool.params.fee` can be set above 100% (1,000,000).
- `MarginlyPool.params.swapFee` can be set above 100% (1,000,000).
- `MarginlyPool.params.positionSlippage` can be set above 100% (1,000,000).
- `MarginlyPool.params.mcSlippage` can be set above 100% (1,000,000).

Recommendation: We recommend adding an upper limit on how high the fees can be.

MAR-23 Clone-and-Own

Informational | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `e79538eefc33ce38f90ae08efabc5250d94fb18b`.
The client provided the following explanation:

```
Fixed as per issue recommendation
```

File(s) affected: `OrableLib.sol`

Description: The clone-and-own approach involves copying and adjusting open-source code at one's discretion. From the development perspective, it is initially beneficial as it reduces the amount of effort. However, from the security perspective, it involves some risks as the code may not follow the best practices, contain a security vulnerability, or include intentionally or unintentionally modified upstream libraries.

`OracleLib.sol` is copied, and its Solidity version is upgraded to `0.8.x`, allowing it to be used with the other contracts. Also, an `unchecked` block is added to the `getSqrtRatioAtTick()` function, as the arithmetic operations were performed without an overflow check in the previous Solidity version.

Recommendation: Rather than the clone-and-own approach, a good industry practice is to use a package manager (e.g., npm) for handling library dependencies. This eliminates the clone-and-own risks yet allows for following best practices, such as, using libraries. If the file is cloned anyway, a comment including the repository, commit hash of the version cloned, and the summary of modifications (if any) should be added. This helps to improve traceability of the file.

MAR-24 Inaccurate Accounting of System Coefficients Due to Precision Loss

Undetermined | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
https://www.overleaf.com/read/bznmrdynygb
```

File(s) affected: `MarginlyPool.sol`

Description: The Marginly pool maintains three system coefficients to track the debt and collateral amount in users' positions. These coefficients include the debt coefficient, collateral coefficient, and deleverage coefficient. Readers can find detailed definitions in the official documentation.

The coefficients used are of type `FP96`, representing fixed-point numbers with 96 bits of precision. The smallest possible value in this representation is `1 / (1 << 96)`, approximately 1.26E-29. In normal circumstances, the debt and collateral coefficients increase as the pool accrues interest from borrowers. However, two coefficients may decrease in the following situations:

- Debt coefficients may decrease when deleveraging occurs.
- Collateral coefficients may decrease when the liquidated position has a negative net difference.

Debt coefficients may decrease significantly when a whale position is liquidated through deleveraging. The size of the liquidated position correlates with the reduction in debt coefficients. In the most extreme case, where the whale's collateral exceeds the total debt of all positions on the opposite side, the debt coefficient can be reduced to only represent the remaining interest portion. As the interest portions are generally much smaller than the total debt in the system, this significantly reduces the debt coefficient.

If whale positions are liquidated through deleveraging multiple times, the updated debt coefficients may eventually fall below the minimum representable value of the FP96 type. However, due to precision loss during the calculations, they will be rounded up to the minimum value. This incorrect updating of debt coefficients can lead to inaccurate internal accounting, potentially impacting user positions.

Exploit Scenario:
Our testing has indicated that debt coefficients may be reduced to the minimum value of the FP96 type, resulting in internal accounting errors. Here is an example scenario:

1. Consider a Marginly ETH/USDC pool with an interest rate of 5%. For simplicity, assume swap and protocol fees are set to 0. The TWAP rate for ETH to USDC is 1 ETH = 4 USDC.
2. User A deposits 7600 USDC into the pool.
3. User B deposits 100 ETH and opens a long position by borrowing 7600 USDC from the pool. This results in a debt of 1900 ETH, with the position at a maximum leverage level of 20.
4. User C deposits 100 USDC and opens a short position by borrowing 100 ETH from the pool.
5. After 10 minutes, The TWAP decreases to 1 ETH = 3.61 USDC, causing User B's position to become unhealthy.
6. User D triggers the `reinit()` function, which liquidates User B's position through deleveraging, as there is insufficient ETH in the pool to swap for USDC.
7. Consequently, the base debt coefficient is reduced to approximately 2.54E-21.
8. If a similar situation occurs as described in Steps 2. to 6. again, the base debt coefficient will be further reduced to the minimum value of type FP96, which is an inaccurate and overestimated value.

Recommendation: Increasing the precision of coefficients could partially mitigate the issue for debt coefficients but may not address it entirely at its root cause.

Collateral coefficients face a similar problem of potentially being reduced to the minimum value of FP96. However, these reductions occur not through deleveraging but due to losses when liquidating positions. This risk can be mitigated by liquidators triggering the liquidation of unhealthy positions before they become insolvent.

Code Documentation

1. Fixed The comment for `baseDelevCoeff` in `MarginlyPool.sol#L73` does not describe the coefficient properly and instead describes the `baseDebtCoeff`.
2. Unresolved In the doc for Pool variables, the calculation of "Base Debt Coef" needs to be corrected to be based on "Short leverage" instead of "Long leverage". The calculation of "Quote Debt Coeff" should be corrected as well.

Adherence to Best Practices

1. Unresolved A call to `MarginlyPool.withdrawQuote()` can be paired with `unwrapWETH = true` parameter. Consider either calling `unwrapAndTransfer()` with a hardcoded value of `false` from this function or revert the call in `unwrapAndTransfer()` if `unwrapWETH = true` and `token != getWETH9Address()`.
2. Fixed `MarginlyRouter` double-inherits `RouterStorage` through the additional inheritance of `AdapterCallback`.
3. Fixed For loops can be gas-optimized by cashing the `array.length` in a `memory` variable and incrementing the iterator via `unchecked {++i;}`.
4. Fixed Remove unused imports:
   - Fixed `MarginlyPool.sol`: `IERC20Minimal.sol`, `AccessControl.sol`
   - Fixed `MarginlyFactory.sol`: `IUniswapV3Pool.sol`
   - Fixed `FullMarginlyFactory.sol`: `IUniswapV3Pool.sol`
   - Fixed `SwapCallback.sol`: `AdapterCallback.sol`
   - Fixed `UniswapV2LikeSwap.sol`: `TransferHelper.sol`
   - Unresolved `UniswapV3LikeSwap.sol`: `SwapCallback.sol`
   - Fixed `WooFiAdapter.sol`: `TransferHelper.sol`
   - Fixed `MarginlyRouter.sol`: `Ownable.sol`, `TransferHelper.sol`, `IMarginlyRouter.sol`, `IMarginlyAdapter.sol`
5. Fixed Remove test code:
   - `MarginlyPool.sol`: `getHeapPosition()`
6. Fixed Consider creating an `onlyOwner` modifier in `MarginlyFactory` and `FullMarginlyFactory` for improved readability. Replace all instances of `if (msg.sender != owner) revert Errors.NotOwner();` with the `onlyOwner` modifier.

```
modifier onlyOwner() {
    if (msg.sender != owner) revert Errors.NotOwner();
}
```

7. Fixed Ensure that `require` statements include an error message to assist with detecting errors. The following functions include `require` statements that are missing error messages:
    1. Fixed `FullMarginlyFactory.createPool()`
    2. Fixed `FullMarginlyFactory.changeSwapRouter()`
    3. Fixed `MarginlyFactory.createPool()`
    4. Fixed `MarginlyFactory.changeSwapRouter()`
8. Fixed Consider indexing the following event arguments to improve log filtering:
    1. `MarginlyKeeper.Profit()`: `liquidatedPosition` and `token`.
9. Unresolved Use the leading underscore naming convention (`_functionName()`) for `private` and `internal` contract functions.
10. Fixed Explicitly set the visibility for contract state variables. The following contract state variables do not have their visibility explicitly set:
    1. `MarginlyPool.SECONDS_IN_YEAR_X96`
    2. `MarginlyPool.UNISWAP_V3_ROUTER_SWAP`
    3. `MarginlyPool.WHOLE_ONE`

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Files

- `246...afc ./packages/router/contracts/MarginlyRouter.sol`
- `588...bba ./packages/router/contracts/interfaces/IMarginlyRouter.sol`
- `fc8...0b8 ./packages/router/contracts/interfaces/IMarginlyAdapter.sol`
- `b18...ba7 ./packages/router/contracts/adapters/UniswapV3Adapter.sol`
- `52c...300 ./packages/router/contracts/adapters/BalancerAdapter.sol`
- `6c6...2ef ./packages/router/contracts/adapters/CamelotAdapter.sol`
- `fb5...62d ./packages/router/contracts/adapters/KyberSwapElasticAdapter.sol`
- `358...c31 ./packages/router/contracts/adapters/UniswapV2Adapter.sol`
- `f37...ef0 ./packages/router/contracts/adapters/WooFiAdapter.sol`
- `288...837 ./packages/router/contracts/adapters/KyberSwapClassicAdapter.sol`
- `95f...b9c ./packages/router/contracts/adapters/ApeSwapAdapter.sol`
- `9e0...683 ./packages/router/contracts/libraries/SwapsDecoder.sol`
- `ace...436 ./packages/router/contracts/abstract/RouterStorage.sol`
- `3bc...37e ./packages/router/contracts/abstract/AdapterCallback.sol`
- `b5e...785 ./packages/router/contracts/abstract/UniswapV2LikeSwap.sol`

- c28...ad0 ./packages/router/contracts/abstract/SwapCallback.sol
- 460...a4d ./packages/router/contracts/abstract/UniswapV3LikeSwap.sol
- 568...449 ./packages/router/contracts/abstract/AdapterStorage.sol
- cc4...ded ./packages/contracts/contracts/FullMarginlyPool.sol
- 699...2ab ./packages/contracts/contracts/MarginlyPool.sol
- 315...f53 ./packages/contracts/contracts/MarginlyKeeper.sol
- 26b...6c3 ./packages/contracts/contracts/MarginlyFactory.sol
- 5c6...eb0 ./packages/contracts/contracts/FullMarginlyFactory.sol
- 01d...da5 ./packages/contracts/contracts/interfaces/IOwnable.sol
- b60...194 ./packages/contracts/contracts/interfaces/IWETH9.sol
- ffb...07c ./packages/contracts/contracts/interfaces/IMarginlyPool.sol
- 463...beb ./packages/contracts/contracts/interfaces/IMarginlyPoolOwnerActions.sol
- 57f...02f ./packages/contracts/contracts/interfaces/IMarginlyFactory.sol
- c0a...c5e ./packages/contracts/contracts/libraries/OracleLib.sol
- ff7...e3e ./packages/contracts/contracts/libraries/MaxBinaryHeapLib.sol
- 087...0f0 ./packages/contracts/contracts/libraries/Errors.sol
- eef...8ae ./packages/contracts/contracts/libraries/FP48.sol
- 134...de9 ./packages/contracts/contracts/libraries/FP96.sol
- 7a0...58a ./packages/contracts/contracts/dataTypes/Mode.sol
- 164...df5 ./packages/contracts/contracts/dataTypes/Position.sol
- 9e2...5a4 ./packages/contracts/contracts/dataTypes/MarginlyParams.sol
- 16b...7ba ./packages/contracts/contracts/dataTypes/Call.sol

Tests

- 65b...6d8 ./int-tests/src/index.ts
- 210...4a3 ./int-tests/src/gen.ts
- 366...c1c ./int-tests/src/contract-api/MarginlyFactory.ts
- d2f...ec7 ./int-tests/src/contract-api/WETH9.ts
- ccc...2ec ./int-tests/src/contract-api/ERC20.ts
- 987...cc8 ./int-tests/src/contract-api/SwapRouter.ts
- 145...051 ./int-tests/src/contract-api/UniswapV3Factory.ts
- 29e...3cd ./int-tests/src/contract-api/UniswapV3Pool.ts
- 2c3...831 ./int-tests/src/contract-api/FiatTokenV2.ts
- 620...63b ./int-tests/src/contract-api/BalancerMarginlyAdapter.ts
- 4a0...dd2 ./int-tests/src/contract-api/MarginlyPool.ts
- c92...a28 ./int-tests/src/contract-api/MarginlyRouter.ts
- ccd...182 ./int-tests/src/contract-api/UniswapV2MarginlyAdapter.ts
- 03c...2fe ./int-tests/src/contract-api/NonfungiblePositionManager.ts
- 02c...ac1 ./int-tests/src/contract-api/MarginlyKeeper.ts
- 117...e8e ./int-tests/src/contract-api/KyberClassicMarginlyAdapter.ts
- 9d1...809 ./int-tests/src/contract-api/UniswapV3MarginlyAdapter.ts
- ea3...889 ./int-tests/src/utils/const.ts
- 58e...0f7 ./int-tests/src/utils/log-utils.ts
- e66...bae ./int-tests/src/utils/chain-ops.ts
- f0d...467 ./int-tests/src/utils/known-contracts.ts
- da3...682 ./int-tests/src/utils/erc20-init.ts
- f88...3d1 ./int-tests/src/utils/types.ts
- f8e...fba ./int-tests/src/utils/logger.ts
- 3c9...ecd ./int-tests/src/utils/fixed-point.ts
- 82d...c58 ./int-tests/src/utils/api-gen.ts
- 858...6f8 ./int-tests/src/utils/uniswap-ops.ts
- 29d...e60 ./int-tests/src/utils/GasReporter.ts
- f41...e93 ./int-tests/src/suites/short_income.ts
- 8f9...c17 ./int-tests/src/suites/short.ts
- d92...b3f ./int-tests/src/suites/long.ts

- `9d6...68b ./int-tests/src/suites/router.ts`
- `3b4...ba7 ./int-tests/src/suites/deleveragePrecision.ts`
- `4dd...ee6 ./int-tests/src/suites/balanceSync.ts`
- `887...44a ./int-tests/src/suites/keeper.ts`
- `219...b55 ./int-tests/src/suites/shutdown.ts`
- `3ae...952 ./int-tests/src/suites/long_income.ts`
- `fc8...ebd ./int-tests/src/suites/index.ts`
- `2dd...9df ./int-tests/src/suites/long_and_short.ts`
- `c1a...142 ./int-tests/src/suites/simulation.ts`
- `650...c5e ./int-tests/contracts/usdc.sol`

Automated Analysis

N/A

Test Suite Results

Independent test suites were provided for the `packages/contracts` and `packages/router` directories. The `packages/contracts` test suite contained 163 tests and the `packages/router` test suite contained 37 tests, all of which have passed.

```
MarginlyFactory
    ✓ should create pool
    ✓ should change router address
    ✓ should raise error when pool exists
    ✓ should raise error when Uniswap pool not found for pair
    ✓ should raise error when trying to create pool with the same tokens

  MarginlyKeeper
    ✓ Should liquidate short bad position
    ✓ Should liquidate long position
    ✓ Should fail when profit after liquidation less than minimum

  Deleverage
    ✓ Deleverage long position
    ✓ Deleverage short position
    ✓ short call after deleverage
    ✓ long call after deleverage
    ✓ depositQuote call after deleverage
    ✓ depositBase call after deleverage
    ✓ withdrawQuote call after deleverage
    ✓ withdrawBase call after deleverage
    ✓ close long position after deleverage
    ✓ close short position after deleverage

MarginlyPool.Liquidation
  ✓ should revert when existing position trying to make liquidation
  ✓ should revert when position to liquidation not exists
  ✓ should revert when position to liquidation not liquidatable
  ✓ should revert when new position after liquidation of short will have bad margin
  ✓ should revert when new position after liquidation of long will have bad margin
  ✓ should create new position without debt after short liquidation
  ✓ should create new position without debt after long liquidation
  ✓ should create new short position after short liquidation
  ✓ should create new long position after long liquidation
  ✓ should create better short position after short liquidation
  ✓ should create better long position after short liquidation

mc heap tests
  ✓ remove long caller
  ✓ remove short caller

MarginlyPool.Shutdown
  ✓ should revert when collateral enough
  ✓ unavailable calls reverted in emergency mode
```

```
    ✓ should switch system in ShortEmergency mode
    ✓ should switch system in LongEmergency mode
pool state after withdraw: base=2 quote=1
    ✓ withdraw tokens for Long/Lend position in ShortEmergency mode
pool state after withdraw: base=1 quote=2
    ✓ withdraw tokens for Short/Lend position in LongEmergency mode
    ✓ should unwrap WETH to ETH when withdraw in Emergency mode
    ✓ should revert withdraw tokens from Short position in ShortEmergency mode
    ✓ should revert withdraw tokens from Long position in LongEmergency mode

  Open position:
    ✓ depositBase
    ✓ depositQuote

  Deposit into existing position:
    ✓ depositBase
    ✓ depositQuote

  System initialized:
    ✓ long
    ✓ short
    ✓ depositBase
    ✓ depositBase and long
    ✓ depositQuote
    ✓ depositQuote and short
    ✓ closePosition
    ✓ withdrawBase
    ✓ withdrawQuote

  mc happens:
    ✓ depositBase with one mc
    ✓ depositQuote with one mc
    ✓ short with one mc
    ✓ long with one mc
    ✓ long initialized heap with one mc
    ✓ long closePosition with one mc
    ✓ depositBase with two mc
    ✓ depositQuote with two mc
    ✓ short with two mc
    ✓ long with two mc
    ✓ closePosition with two mc
    ✓ MC long position with deleverage
    ✓ MC short position with deleverage
    ✓ MC long reinit
    ✓ MC short reinit

  Liquidation
    ✓ liquidate long position and create new position
    ✓ liquidate short position and create new position
    ✓ liquidate long position and create new long position
    ✓ liquidate short position and create new short position

  MarginlyPool.Base
    ✓ should revert when second try of initialization
    ✓ should revert when somebody trying to send value
    ✓ sweepETH should revert when sender is not admin
    ✓ sweepETH should be called by admin
    ✓ should set Marginly parameters by factory owner
    ✓ should raise error when not an owner trying to set parameters
    ✓ should limit system leverage after long liquidation
    ✓ should limit system leverage after short liquidation
    Deposit base
      ✓ zero amount
      ✓ exceeds limit
      ✓ first deposit should create position
      ✓ different signers deposits
      ✓ deposit into positive base position
      ✓ depositBase into short position
      ✓ depositBase into long position
      ✓ depositBase and open long position
      ✓ depositBase and long into short position should fail
      ✓ depositBase should wrap ETH into WETH
```

```
Deposit quote
  ✓ zero amount
  ✓ exceeds limit
  ✓ first deposit should create position
  ✓ deposit into positive quote position
  ✓ deposit into short position
  ✓ deposit into long position
  ✓ depositQuote and open short position
  ✓ depositQuote and short into long position
  ✓ depositQuote and short into short position
  ✓ depositQuote should wrap ETH to WETH
Withdraw base
  ✓ should raise error when trying to withdraw zero amount
  ✓ should raise error when position not initialized
  ✓ should decrease base position
  ✓ withdraw with position removing
  ✓ withdrawBase should unwrap WETH to ETH
  ✓ should raise error when trying to withdraw from short position
Withdraw quote
  ✓ should raise error when trying to withdraw zero amount
  ✓ should raise error when position not initialized
  ✓ should decrease quote position
  ✓ reinit
  ✓ withdraw with position removing
  ✓ withdrawQuote should unwrap WETH to ETH
  ✓ should raise error when trying to withdraw from long position
Close position
  ✓ should raise error when attempt to close Uninitialized or Lend position
  ✓ close short slippage fail
  ✓ close long slippage fail
  ✓ should close short position
  ✓ should close long position
Short
  ✓ short, wrong user type
  ✓ short minAmount violation
  ✓ exceeds limit
  ✓ slippage fail
  ✓ should not exceed quoteLimit when deposit base cover debt
  ✓ could exceed quoteLimit when deposit base amount
  ✓ short should update leverageShort
  ✓ short, changed from lend to short
  ✓ short, update short position
Long
  ✓ uninitialized
  ✓ long minAmount violation
  ✓ exceeds limit
  ✓ slippage fail
  ✓ should not exceed quoteLimit when deposit base cover debt
  ✓ could exceed quoteLimit when deposit quote amount
  ✓ long should update leverageLong
  ✓ changed from lend to long
  ✓ update long position
 Position sort keys
  ✓ should properly calculate sort key for long position
  ✓ should properly calculate sort key for short position
  ✓ long position sortKey
  ✓ short position sortKey

MaxBinaryHeapTest
  ✓ should create empty heap
  ✓ should return false when peek root on empty heap
  ✓ should return isEmpty
  ✓ should add new item and rebuild tree
  ✓ should return success=false when trying to get index of not existed in the heap account
  ✓ should return success=false on empty heap
  ✓ should remove root item and rebuild tree
  ✓ should remove item by index
  ✓ should remove from last to top
  ✓ should remove arbitary element
  ✓ should remove element and update tree (heapifyUp)
  ✓ should remove element and update tree (heapifyDown)
  ✓ should create max binary heap and remove items in right order
```

```
    Should update heap by index
      ✓ without changing position
      ✓ from middle to top
      ✓ from middle to bottom
      ✓ from top to bottom
      ✓ from bottom to top
      ✓ should update node account by index

  TestSwapRouter
    - swap base to quote exact input
    - swap quote to base exact input
    - swap base to quote exact output
    - swap quote to base exact output

  159 passing (1m)
  4 pending
```

```
MarginlyRouter UniswapV3
    ✔ swapExactInput 0 to 1, success (2277ms)
    ✔ swapExactInput 0 to 1, less than minimal amount (78ms)
    ✔ swapExactInput 1 to 0, success (102ms)
    ✔ swapExactInput 1 to 0, less than minimal amount (80ms)
    ✔ swapExactOutput 0 to 1, success (98ms)
    ✔ swapExactOutput 0 to 1, more than maximal amount (75ms)
    ✔ swapExactOutput 1 to 0, success (96ms)
    ✔ swapExactOutput 1 to 0, more than maximal amount (69ms)

  MarginlyRouter UniswapV2
    ✔ swapExactInput 0 to 1, success (89ms)
    ✔ swapExactInput 0 to 1, less than minimal amount (46ms)
    ✔ swapExactInput 1 to 0, success (87ms)
    ✔ swapExactInput 1 to 0, less than minimal amount (45ms)
    ✔ swapExactOutput 0 to 1, success (92ms)
    ✔ swapExactOutput 0 to 1, more than maximal amount (46ms)
    ✔ swapExactOutput 1 to 0, success (91ms)
    ✔ swapExactOutput 1 to 0, more than maximal amount (51ms)

  MarginlyRouter Balancer Vault
    ✔ swapExactInput 0 to 1, success (98ms)
    ✔ swapExactInput 0 to 1, less than minimal amount (74ms)
    ✔ swapExactInput 1 to 0, success (90ms)
    ✔ swapExactInput 1 to 0, less than minimal amount (60ms)
    ✔ swapExactOutput 0 to 1, success (95ms)
    ✔ swapExactOutput 0 to 1, more than maximal amount (59ms)
    ✔ swapExactOutput 1 to 0, success (107ms)
    ✔ swapExactOutput 1 to 0, more than maximal amount (66ms)

  MarginlyRouter WooFi
    ✔ swapExactInput 0 to 1, success (97ms)
    ✔ swapExactInput 0 to 1, less than minimal amount (71ms)
    ✔ swapExactInput 1 to 0, success (83ms)
    ✔ swapExactInput 1 to 0, less than minimal amount (66ms)
    ✔ swapExactOutput error (57ms)

  Callbacks
    ✔ adapter callback fails if sender is unknown
    ✔ uniswapV3 callback fails if sender is unknown

  SwapInfo decoding
    ✔ default
    ✔ only uniswapV3
[ 1, 9 ]
[ 32671, 97 ]
    ✔ split randomly between 2 Dexs
[ 9, 5 ]
[ 16984, 12860 ]
    ✔ Wrong swap ratios
[ 8, 0 ]
[ 2217, 30551 ]
    ✔ Wrong swaps number
```

```
[ 0 ]
    ✔ WrongSwapsNumber: not zero in the end


  37 passing (5s)
```

Code Coverage

The tests provided were run successfully. The code coverage for the `packages/contracts` and `packages/router` directories is calculated separately. For the `packages/contracts` directory, statement coverage is 94%, and branch coverage is 80%. As for the `packages/router` directory, statement coverage is 68%, and branch coverage is 50%. In both directories, some contracts remain completely untested (`FullMarginlyFactory.sol`, `FullMarginlyPool.sol`, `ApeSwapAdapter.sol`, `CamelotAdapter.sol`, `KyberSwapClassicAdapter.sol`, and `KyberSwapElasticAdapter.sol`).

We highly recommend adding tests to cover all contracts and reach 90% and higher branch- and statement-coverage.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	95.32	80.52	91.78	93.24	
FullMarginlyFactory.sol	0	0	0	0	... 74,79,80,81
FullMarginlyPool.sol	0	0	0	0	... 46,47,48,49
MarginlyFactory.sol	100	75	100	100	
MarginlyKeeper.sol	96.97	66.67	100	97.22	107
MarginlyPool.sol	97.6	85.56	100	97.15	... 6,1467,1468
contracts/dataTypes/	100	100	100	100	
Call.sol	100	100	100	100	
MarginlyParams.sol	100	100	100	100	
Mode.sol	100	100	100	100	
Position.sol	100	100	100	100	
contracts/interfaces/	100	100	100	100	
IMarginlyFactory.sol	100	100	100	100	
IMarginlyPool.sol	100	100	100	100	
IMarginlyPoolOwnerActions.sol	100	100	100	100	
IWETH9.sol	100	100	100	100	
contracts/libraries/	90	80	68.97	89.15	
Errors.sol	100	100	100	100	
FP48.sol	100	100	100	100	
FP96.sol	50	50	52.63	60	... 127,131,159

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
MaxBinaryHeapLib.sol	100	91.67	100	100	
OracleLib.sol	100	78	100	100	
All files	94.55	80.41	85.29	92.58	

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	50	100	100	
MarginlyRouter.sol	100	50	100	100	
contracts/abstract/	80.95	64.29	84.62	73.85	
AdapterCallback.sol	100	100	100	100	
AdapterStorage.sol	55.56	25	66.67	55	... 43,44,46,47
RouterStorage.sol	63.64	25	66.67	57.89	... 41,42,44,45
SwapCallback.sol	100	70	100	100	
UniswapV2LikeSwap.sol	100	83.33	100	100	
UniswapV3LikeSwap.sol	100	100	100	100	
contracts/adapters/	47.5	29.17	44.83	52.14	
ApeSwapAdapter.sol	0	0	0	0	... 37,38,39,40
BalancerAdapter.sol	100	50	100	100	
CamelotAdapter.sol	0	0	0	0	... 21,22,23,34
KyberSwapClassicAdapter.sol	0	0	0	0	... 71,72,73,74
KyberSwapElasticAdapter.sol	0	0	0	0	... 76,78,86,92
UniswapV2Adapter.sol	100	100	100	100	
UniswapV3Adapter.sol	100	70	100	100	
WooFiAdapter.sol	100	50	100	100	
contracts/interfaces/	100	100	100	100	
IMarginlyAdapter.sol	100	100	100	100	
IMarginlyRouter.sol	100	100	100	100	
contracts/libraries/	100	100	100	100	

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
SwapsDecoder.sol	100	100	100	100	
All files	68.55	50	60.87	68.94	

Changelog

- 2023-09-15 - Initial report
- 2023-10-09 - First Fix Review
- 2023-10-11 - Final Report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

unknown risks and flaws and, as such, the report may not be complete or inclusive of all vulnerabilities. The review is limited to the materials identified in the report and does not extend to the compiler layer, or any other areas beyond the programming language, or programming aspects that could present security risks. The report does not indicate the endorsement by Quantstamp of any particular project or team, nor guarantee its security, and and may not be represented as such. No third party is entitled to rely on the report in any any way, including for the purpose of making any decisions to buy or sell a product, product, service or any other asset. Quantstamp does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or or any open source or third-party software, code, libraries, materials, or information to, to, called by, referenced by or accessible through the report, its content, or any related related services and products, any hyperlinked websites, or any other websites or mobile applications, and we will not be a party to or in any way be responsible for monitoring any any transaction between you and any third party. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.