Code Security Assessment

Kromatika.Finance

Feb 10th, 2022

Table of Contents

Summary

This report has been prepared for Kromatika.Finance to discover issues and vulnerabilities in the source code of the Kromatika.Finance project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

Overview

Project Summary

Project Name	Kromatika.Finance
Description	ERC20
Platform	Ethereum
Language	Solidity
Codebase	https://github.com/Kromatika-Finance/limit-trade
Commit	

Audit Summary

Delivery Date	Feb 10, 2022
Audit Methodology	Static Analysis, Manual Review

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Mitigated	Resolved
Critical	1	0	0	0	0	0	1
Major	5	0	0	0	2	1	2
Medium	1	0	0	0	0	0	1
Minor	5	0	0	3	0	0	2
Informational	16	0	0	0	1	0	15
Discussion	0	0	0	0	0	0	0

Audit Scope

ID	File	SHA256 Checksum
IOM	interfaces/IOrderManager.sol	1d5d6401b3495196567c087e76c3d1f78d62160abd65b7b14a55d1ae51404112
IOC	interfaces/IOrderMonitor.sol	07e94691c24eebb7ed8c4597145497dce8d3f6d00628075cafa60f6829bedac3
KCK	Kromatika.sol	f71055085b3b2897162d52f7ff4303895ece2a3d3b02bd83b7c1d1881eec0089
LOM	LimitOrderManager.sol	6bb817a6b9db2cc06c7b4304df34bc1958e00b0698b066f6adcc2eaf8e20ad6e
LOC	LimitOrderMonitor.sol	2ee9290331154 78e8f7b18484a601b0dcbdd3efca5ce913069352dfb09559239
LOK	LimitOrderMonitorChainlink.sol	17007360aa95f8c008ae9bb20d1bb64252aec0654228091ee8e71067419cb744
MCK	Multicall.sol	8cef9083060ace60ade8d906612c00107ce4c7745c7987df7f4ace64c7afee6c
REA	README.md	11f5e6619d80dfb4297702c67ff64d025e707d6e4092a25e972626f92976d314
SPC	SelfPermit.sol	5535333864d2ff5d583555f0293c9ad0eb1c6784e2303490500e201b5e05ec8e
UUC	UniswapUtils.sol	46386874ece66de48f1ff839a2975829dd0ed7ec47b94559475b01c10355d62f

Understandings

Overview

Dependencies

There are a few depending injection contracts or addresses in the current project:

- `factory`, `WETH`, `KROM`, `IUniswapV3Pool(limitOrder.pool)`, and `limitOrder.monitor` for contract `LimitOrderManager`;
- `orderManager`, `factory`, and `KROM` for the contract `LimitOrderMonitor`;
- `orderManager`, `factory`, `KROM`, `swapRouter`, `LINK`, and `WETH` for the contract `LimitOrderMonitorChainlink`.

We assume these contracts or addresses are valid and non-vulnerable actors and implement proper logic to collaborate with the current project.

Privilledged Functions

In the contract `LimitOrderManager`, the role `owner()` has the authority over the following functions:

- `setMonitors()`
- `setMarginGasUsageMultiplier()` The role `monitor` has the authority over the following functions:
- `processLimirOrder()`

In the contract `LimitOrderMonitor`, the role `owner()` has the authority over the following functions:

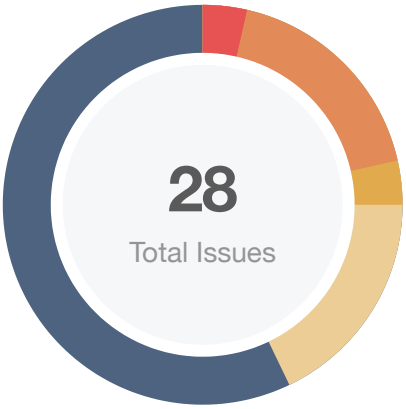
- `setBatchSize()`
- `setMonitorSize()`
- `setUpkeepInterval()`
- `setKeeperFee()` The role `orderManager` has the authority over the following functions:
- `startMonitor()`
- `stopMonitor()`

In the contract `LimitOrderMonitorChainlink`, the role `owner()` has the authority over the following function:

- `setKeeperId()`

To improve the trustworthiness of the project, any dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of the `Timelock` contract.

Findings



Critical	**1** (3.57%)
Major	**5** (17.86%)
Medium	**1** (3.57%)
Minor	**5** (17.86%)
Informational	**16** (57.14%)
Discussion	**0** (0.00%)

ID	Title	Category	Severity	Status
KROM-01	Centralization Risk	**Centralization / Privilege**	**Major**	Partially Resolved
KROM-02	Third Party Dependencies	Volatile Code	Minor	Acknowledged
KROM-03	Missing Emit Events	Coding Style	Informational	Resolved
KROM-04	Return Value Ignored	Volatile Code	Informational	Resolved
KROM-05	Unlocked Compiler Version	Language Specific	Informational	Resolved
KROM-06	Initialize Functions Lack Restrictions	Logical Issue	Informational	Partially Resolved
KCK-01	Centralization Related Risks	**Centralization / Privilege**	**Major**	Mitigated
LOC-01	Missing Requirement	Logical Issue	Informational	Resolved
LOC-02	Missing Error Messages	Coding Style	Informational	Resolved
LOC-03	Missing Input Validations	Volatile Code	Medium	Resolved
LOK-01	Missing Error Messages	Coding Style	Informational	Resolved
LOK-02	Potential Sandwich Attacks	Logical Issue	Major	Resolved
LOK-03	Unsafe Integer Cast	Logical Issue	Minor	Resolved
LOM-01	Missing Error Messages	Coding Style	Informational	Resolved

ID	Title	Category	Severity	Status
LOM-02	Critical State Variable Not Updated When Transferring ERC721 Tokens	Logical Issue	Critical	Resolved
LOM-03	Incorrect `require` Statement	Logical Issue	Minor	Resolved
LOM-04	Proper Usage of `require` And `assert` Functions	Coding Style	Informational	Resolved
LOM-05	Proper `monitors` Initialization	Logical Issue	Informational	Resolved
LOM-06	Unsafe Implicit Integer Casting	Volatile Code	Informational	Resolved
LOM-07	Integer Overflow Risk	Mathematical Operations	Informational	Resolved
LOM-08	Not Collecting Fees Earned in Uniswap V3 Pools	Logical Issue	Minor	Acknowledged
UUC-01	Missing Error Messages	Coding Style	Informational	Resolved
UUC-02	Potential Oracle Manipulation	Logical Issue	Major	Partially Resolved
UUC-03	Potential Price Manipulation	Logical Issue	Major	Resolved
UUC-04	Assumption of Possitive Tick Spacing	Logical Issue	Minor	Acknowledged
UUC-05	Redundant `import` Files	Coding Style	Informational	Resolved
UUC-06	Integer Overflow/Underflow Risk	Mathematical Operations	Informational	Resolved
UUC-07	Proper Usage of `require` And `assert` Functions	Coding Style	Informational	Resolved

KROM-01 | Centralization Risk

Category	Severity	Location	Status
Centralization / Privilege	Major	Global	Partially Resolved

Description

In the contract `LimitOrderManager`, the role `owner()` has the authority over the following functions:

- `setMonitors()`
- `setMarginGasUsageMultiplier()`

In the contract `LimitOrderMonitor`, the role `owner()` has the authority over the following functions:

- `setBatchSize()`
- `setMonitorSize()`
- `setUpkeepInterval()`
- `setKeeperFee()`

In the contract `LimitOrderMonitor`, the role `orderManager` has the authority over the following functions:

- `startMonitor()`
- `stopMonitor()`

In the contract `LimitOrderMonitorChainlink`, the role `owner()` has the authority over the following function:

- `setKeeperId()`

Any compromise to the `owner()` or `orderManager` accounts may allow the hacker to take advantage of these functions.

Recommendation

We advise the client to carefully manage the `owner()` and `orderManager` accounts's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol to be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., Multisignature wallets.

Indicatively, here are some suggestions that would mitigate the potential risk in the short-term and long-term:

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;

- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key;
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.

Alleviation

[Kromatika Team]: Kromatika uses a Gnosis safe 3-4 multisignature wallet that will hold the privileged role to change the parameters of the corresponding smart contracts in a decentralized way. Later on Kromatika plans on introducing a governance module to transfer the privileges to the DAO.

KROM-02 | Third Party Dependencies

Category	Severity	Location	Status
Volatile Code	Minor	Global	Acknowledged

Description

The contract is serving as the underlying entity to interact with third party UniSwap V3 and Chainlink protocols. The scope of the audit treats 3rd party entities as black boxes and assume their functional correctness. However, in the real world, 3rd parties can be compromised and this may lead to lost or stolen assets. In addition, upgrades of 3rd parties can possibly create severe impacts, such as increasing fees of 3rd parties, migrating to new LP pools, etc.

Recommendation

We understand that the business logic of Kromatika Finance requires interaction with UniSwap and Chainlink. We encourage the team to constantly monitor the statuses of 3rd parties to mitigate the side effects when unexpected activities are observed.

Alleviation

[Kromatika Team]: Kromatika will use a third party monitoring service (OpenZepellin Defender) for monitoring theactivities of 3rd parties.

KROM-03 | Missing Emit Events

Category	Severity	Location	Status
Coding Style	Informational	Global	Resolved

Description

The function that affects the status of sensitive variables should be able to emit events as notifications. In the contract `LimitOrderManager`:

- `initialize()` which sets `marginGasUsageMultiplier`
- `setMarginGasUsageMultiplier()` which sets `marginGasUsageMultiplier`

In the contract `LimitOrderMonitor`:

- `setBatchSize()` which sets `batchSize`
- `setMonitorSize()` which sets `monitorSize`
- `setUpkeepInterval()` which sets `upkeepInterval`
- `setKeeperFee()` which sets `monitorFee`
- `startMonitor()` which starts monitor for a tokenID
- `stopMonitor` which stops monitor for a tokenID

In the contract `LimitOrderMonitorChainlink`:

- `setKeeperId()` which sets `keeperId`

Recommendation

Consider adding events for sensitive actions, and emit them in the function.

Alleviation

[Kromatika Team]: he team has applied a fix for this issue: https://github.com/Kromatika-Finance/limit-trade/commit/1ea8112beb6663dc69d841e846c95d118043939b

The following events have been added:

- LimitOrderManager
- GasUsageMonitorChangedLimitOrderMonitor
- BatchSizeChanged
- MonitorSizeChanged

- UpkeepIntervalChanged
- MonitorStarted
- MonitorStopped

KROM-04 | Return Value Ignored

Category	Severity	Location	Status
Volatile Code	Informational	Global	Resolved

Description

In the contract `LimitOrderManager`, the functions `WETH.transfer()`, `_pool.mint()`, `_collect()` and `CallbackValidation.verifyCallback()`, and in the contract `LimitOrderMonitor` the function `KROM.approve()` invocations do not check the return value of the function call which should yield respective values in case of a proper call.

Recommendation

We recommend adding appropriate return value checks to ensure that the function calls are successful.

Alleviation

[Kromatika Team]: The team has applied a fix for this issue: https://github.com/Kromatika-Finance/limit-trade/commit/f4f3fe05ee957e0926e18f94924be0db6def04c5All the recommended possible checks have been validated with require call.

KROM-05 | Unlocked Compiler Version

Category	Severity	Location	Status
Language Specific	Informational	Global	Resolved

Description

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.2` the contract should contain the following line:

```
pragma solidity 0.6.2;
```

Alleviation

[Kromatika Team]: With the fix commit: 9d0e681aec381aef07b1e09fd7b1ed25db96b595all smart contracts have been locked to using compiler version `pragma solidity 0.7.6;` Additionally, Kromatika uses truffle framework that locks the solidity compiler to version 0.7.6 (in file truffle-config.js)

KROM-06 | Initialize Functions Lack Restrictions

Category	Severity	Location	Status
Logical Issue	Informational	Global	Partially Resolved

Description

The following contracts have configure or initialize functions that can be called by anyone:

- `LimitOrderManager`
- `LimitOrderMonitor`
- `LimitOrderMonitorChainlink`

This allows attackers to potentially front-run initialization transactions and manipulate sensitive variables.

Recommendation

Consider placing restrictions on who is able to properly initialize these contracts.

Alleviation

[Kromatika Team]: Kromatika uses an OpenZepellin TransparentUpgradableProxy that creates a proxy contract for the logic contracts:

- LimitOrderManager
- LimitOrderMonitor
- LimitOrderMonitorChainlink

UpgradableProxy initializes the proxy contract within the same transaction in the constructor. Thus, the initialization transaction happens within the same proxy contract deployment transaction and can be executed only once. Kromatika is using OpenZepellin truffle upgrades plugin for creating upgradable proxies with initialization logic.

Update (Jan31 th 2022): Kromatika team with the commit: 344165e12eb1e0c9a60781a7984e061dc3867d02has also added a constructor for its implementations contracts to automatically mark them as initialized when deployed, so that the initialize() method cannot be called again

KCK-01 | Centralization Related Risks

Category	Severity	Location	Status
Centralization / Privilege	**Major**	Kromatika.sol: 11	Mitigated

Description

All of the KROM tokens are sent to the contract deployer when deploying the contract. This could be a centralization risk as the deployer can distribute KROM tokens without obtaining the consensus of the community.

Recommendation

We recommend the team to be transparent regarding the initial token distribution process, and the team shall make enough efforts to restrict the access of the private key.

Alleviation

[Kromatika Team]: Immediately after the KROM token distribution, 20% of the KROM token has been locked in the Gnosis safe 3-4 multisignature wallet and 80% distributed on DEXes as initial dex liquidity. The deployer account does not keep any of the initial KROM token distribution

LOC-01 | Missing Requirement

Category	Severity	Location	Status
Logical Issue	Informational	LimitOrderMonitor.sol: 160	Resolved

Description

In the contract `LimitOrderMonitor`, the function `performUpkeep` unpacks an uint array `_tokenIds` and uint `count` from the input data but does not verify that the `_tokenIds.length` is greater than or equal to `count` to ensure that the array can be properly accessed when looping through.

Recommendation

We recommend including the following check:

```
require(_count =< _tokenIds.length);
```

Alleviation

[Kromatika Team]: The team has applied a fix for this issue: https://github.com/Kromatika-Finance/limit-trade/commit/70b2706e9646179d58cb7fa100d6f669631719f9

Adding line

```
require(_count <= _tokenIds.length);
```

LOC-02 | Missing Error Messages

Category	Severity	Location	Status
Coding Style	Informational	LimitOrderMonitor.sol: 179, 185	Resolved

Description

The **require** can be used to check for conditions and throw an exception if the condition is not met. It is better to provide a string message containing details about the error that will be passed back to the caller.

Recommendation

We advise adding error messages to the linked **require** statements.

Alleviation

[Kromatika Team]: With the fix commit: b67ce36a150adaaa992e6810cd4bb84272552e46, Kromatika team have added short error messages to all require statements

LOC-03 | Missing Input Validations

Category	Severity	Location	Status
Volatile Code	Medium	LimitOrderMonitor.sol: 149~151	Resolved

Description

As pointed out by the Chainlink team from the chainlink github, the input to the function `LimitOrderMonitor.performUpdate` should not be trusted, and the caller of the method should not even be restricted to any single registry. Anyone should be able to call it, and the input should be validated, there is no guarantee that the data passed in is the `performData` returned from `LimitOrderMonitor.checkUpkeep()`.

This could happen due to malicious keepers, racing keepers, or simply a state change while the `LimitOrderMonitor.performUpkeep()` transaction is waiting for confirmation.

For example, a malicious keeper may simply guess a few `_tokenId` and pass them into the `LimitOrderMonitor.performUpkeep()` function. If at least one of the `_tokenId` indeed needs to be up-kept, this function would finish successfully, and `lastUpkeep` will be updated. Then, the good keepers will not be able to perform the right `LimitOrderMonitor.checkUpkeep()` due to the fact that `lastUpkeep == block.number` and further they can't pass in the right input into `LimitOrderMonitor.performUpkeep()` function.

Recommendation

The audit team advise adding enough validations for the passed-in values to prevent unexpected errors.

Alleviation

[Kromatika Team]: With the commit fix: 4099bf1f630dcbc9d1a2784e4726ae98b465a57f Kromatika team has added additional input validation check in regarding the `_tokenIds` and `_count` checks related to `batchSize`. In order to prevent malicious keepers guessing `_tokenId`, with commit: b9ffed3cf2868705a53072a90bb8a0729f8e7380, Kromatika team has also removed the `lastUpkeep` and will check for upkeeps every block

LOK-01 | Missing Error Messages

Category	Severity	Location	Status
Coding Style	Informational	LimitOrderMonitorChainlink.sol: 58	Resolved

Description

The **require** can be used to check for conditions and throw an exception if the condition is not met. It is better to provide a string message containing details about the error that will be passed back to the caller.

Recommendation

We advise adding error messages to the linked **require** statements.

Alleviation

[Kromatika Team]: With the fix commit: b67ce36a150adaaa992e6810cd4bb84272552e46, Kromatika team have added short error messages to all require statements

LOK-02 | Potential Sandwich Attacks

Category	Severity	Location	Status
Logical Issue	Major	LimitOrderMonitorChainlink.sol: 62~72	Resolved

Description

A sandwich attack might happen when an attacker observes a transaction swapping tokens or adding liquidity without setting restrictions on slippage or minimum output amount. The attacker can manipulate the exchange rate by frontrunning (before the transaction being attacked) a transaction to purchase one of the assets and make profits by backrunning (after the transaction being attacked) a transaction to sell the asset.

The following functions are called without setting restrictions on slippage or minimum output amount, so transactions triggering these functions are vulnerable to sandwich attacks, especially when the input amount is large:

```
62              ISwapRouter.ExactInputParams memory params =
63              ISwapRouter.ExactInputParams({
64                  path: abi.encodePacked(address(KROM), POOL_FEE, address(WETH),
POOL_FEE, address(LINK)),
65                  recipient: address(this),
66                  deadline: block.timestamp,
67                  amountIn: _amount,
68                  amountOutMinimum: 0
69              });
70
71              // swap and send
72              _amount = swapRouter.exactInput(params);
```

Recommendation

We recommend setting reasonable minimum output amounts, instead of 0, based on token prices when calling the aforementioned functions.

Alleviation

[Kromatika Team]: With the fix commit: 18f7694701e81a5571c78659b1092dc2d4855b60, Kromatika team has removed the sandwich attack code considering it as an obsolete / unnecessary feature. The swapping and replenishment of Chainlink keepers will be done by an Externally Owned Account(EOA) managed automatically by a third party OpenZepellin Depender.

LOK-03 | Unsafe Integer Cast

Category	Severity	Location	Status
Logical Issue	Minor	LimitOrderMonitorChainlink.sol: 75	Resolved

Description

The linked statements cast a `uint256` value to an `uint96` without evaluating its bounds.

Recommendation

The audit team advise a safe casting operation to be performed by ensuring the result is correct.

Alleviation

[Kromatika Team]: With the fix commit: 18f7694701e81a5571c78659b1092dc2d4855b60, Kromatika team has removed code that was containing the unsafe cast operation.

LOM-01 | Missing Error Messages

Category	Severity	Location	Status
Coding Style	Informational	LimitOrderManager.sol: 159, 188, 189, 210, 238, 280, 293, 331, 405, 492, 505, 542, 554, 609	Resolved

Description

The **require** can be used to check for conditions and throw an exception if the condition is not met. It is better to provide a string message containing details about the error that will be passed back to the caller.

Recommendation

We advise adding error messages to the linked **require** statements.

Alleviation

[Kromatika Team]: With the fix commit: b67ce36a150adaaa992e6810cd4bb84272552e46, Kromatika team have added short error messages to all require statements

LOM-02 | Critical State Variable Not Updated When Transferring ERC721 Tokens

Category	Severity	Location	Status
Logical Issue	Critical	LimitOrderManager.sol: 26~32	Resolved

Description

The `LimitOrderManager` is an ERC721Upgradeable which mints ERC721 tokens to represent limit orders. Since the ERC721 tokens are transferrable, users may transfer their tokens to others. However, when transferring the tokens, the critical state variable `activeOrders` is not updated accordingly. The `activeOrder` is a mapping from the token owner's address to the number of active orders the owner has.

Without correctly updating this critical variable `activeOrders`, the following consequences may happen:

- `processLimitOrder()` may experience denial of service due to the failure of SafeMath operations on line 222 `activeOrders[_owner] = activeOrders[_owner].sub(1)`;
- the new token owner may not be able to cancel the limit in `cancelLimitOrder()` due to the failure of SafeMath operations on line 255;
- the fee calculated by the function `estimateServiceFee(targetGasPrice[msg.sender], activeOrders[msg.sender])` would be incorrect;
- the function `isUnderfunded` may return an incorrect boolean because of the incorrect result from `estimateServiceFee()`;
- `_createLimitOrder` would also be affected by erroneously setting the `activeOrders`.

Recommendation

The audit team recommend updating the aforementioned the state variable when transferring ERC721 tokens.

Alleviation

[Kromatika Finance]: With the fix commit: [48eb64cedb84322f79365e253f275682fedb5481] Kromatika team has fixed the issue. Now the `activeOrders` are updated whenever there is a valid transfer of the NFT `tokenId` between `from` and `to` address and the `tokenId` is still active i.e limit order represented by that token is still not processed.

LOM-03 | Incorrect `require` Statement

Category	Severity	Location	Status
Logical Issue	Minor	LimitOrderManager.sol: 293	Resolved

Description

In the contract `LimitOrderManager`, the function `withdrawFunding()` checks to ensure that the account has enough `funding` balance to satisfy `reservedServiceFee`. However, the `require` statement checks the balance without accounting for the withdrawal amount, resulting in an account potentially unable to satisfy the `reservedServiceFee`.

Recommendation

We recommend the following requirement:

```
require(balance - _amount >= reservedServiceFee);
```

Alleviation

[Kromatika Team]: This is a logical issue and we have removed the reserved service fees concept completely, making this issue non relevant. The users will no longer have a reserved service fee and they can withdraw any `_amount` up to the `balance`, making this check unnecessary.
https://github.com/Kromatika-Finance/limit-trade/blob/1ea8112beb6663dc69d841e846c95d118043939b/contracts/LimitOrderManager.sol

LOM-04 | Proper Usage Of `require` And `assert` Functions

Category	Severity	Location	Status
Coding Style	Informational	LimitOrderManager.sol: 554	Resolved

Description

The `assert()` function should only be used to test for internal errors, and to check invariants since the gas fee will not be refunded. The `require()` function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts.

Recommendation

Consider using the `require()` function, along with a custom error message when the condition fails, instead of the `assert` function.

Alleviation

[Kromatika Team]: With the fix commit: 66e112989600f8ff0b08dbb05de9c193d3ecf02f, Kromatika team has replaced `assert()` with `require()` function with a custom message.

LOM-05 | Proper `monitors` Initialization

Category	Severity	Location	Status
Logical Issue	Informational	LimitOrderManager.sol: 542	Resolved

Description

When users create limit order, the corresponding monitor needs to be selected. In the function `LimitOrderManager._selectMonitor()`, it requires the `monitors.length > 0`.

```
541         uint256 monitorLength = monitors.length;
542         require(monitorLength > 0);
```

However, the `monitors` is not properly initialized in the `initialize()` but only in the `LimitOrderManager.setMonitors()`. In the case of improper initialization, users may experience Denial of Service.

Recommendation

The audit team recommend ensuring the proper initialization of the `monitors`.

Alleviation

[Kromatika Team]: With the fix commit: f1779c8dd6f824425062c003bfbd70f993748d4d, Kromatika team has initialized the monitors within the contract initialization function.

LOM-06 | Unsafe Implicit Integer Casting

Category	Severity	Location	Status
Volatile Code	Informational	LimitOrderManager.sol: 175	Resolved

Description

The `nextId` is type `uint176` and `_tokenId` is `uint256`, and an implicit integer cast is accomplished on line 175.

```
175         _mint(msg.sender, (_tokenId = nextId++));
```

The operation is supported in this version of Solidity, but may not be supported in a future version, and using implicit integer casting may make the code more prone to errors.

Recommendation

The audit team recommend using explicit integer casting or declare the `nextId` to be `uint256`.

Alleviation

[Kromatika Finance]: With the commits: 348bcfe15f0d56710770d5c57d5efd561bf59d17 and c331abb74acc96fd67521aee61a9d4965b060e26, the team has updated the `nextId` to uint256.

LOM-07 | Integer Overflow Risk

Category	Severity	Location	Status
Mathematical Operations	Informational	LimitOrderManager.sol: 175	Resolved

Description

Using `+` in the method directly to calculate the value of the variable may overflow. `SafeMath` provides a method to verify overflow, and it is safer to use the method provided.

Recommendation

Consider using the `add()` function in `SafeMath` library for mathematical operations.

Alleviation

[Kromatika Team]: With the fix commits: 361742b7a88f211e5bd2a5e2a76a0dd3ab2eae87, Kromatika team has introduced `SafeCast` and `SafeMath` libraries from OpenZepellin for safe mathematical and casting operations over uint256, uin128 and uint32 types used in the contract.

LOM-08 | Not Collecting Fees Earned In Uniswap V3 Pools

Category	Severity	Location	Status
Logical Issue	Minor	LimitOrderManager.sol: 265	Acknowledged

Description

When users burn their limit order token and collect the corresponding token pairs, the function `LimitOrderManager.collect()` is called, which further calls the function `pool.collect()`. However, `pool.collect()` does not recompute fees earned, which must be done either via mint or burn of any amount of liquidity. In contrast, users can call `LimitOrderManager.cancelLimitOrder()` to cancel their order and withdraw their tokens where the burn of liquidity is done via `pool.burn()` that updates the pool status, adding the fees to the number of collectible tokens. We would like to discuss with the team whether the logic of the function `LimitOrderManager.collect()` reflects the intended design.

Recommendation

We recommend adding features to collect the proportionate fees.

Alleviation

[Kromatika Team]: Kromatika team is acknowledging the recommendation and would like to describe the solution. In the method `LimitOrderManager._removeLiquidity()`, Kromatika calculates the `tokensOwed0` and `tokensOwed1` that represents the tokens amount owed to the user from Uniswap pool, including the fees by using the fee growth from Uniswappool:

```
(, uint256 feeGrowthInside0LastX128, uint256 feeGrowthInside1LastX128, , )
=_pool.positions(positionKey);
```

`tokensOwed0` and `tokensOwed1` are then passed to the `internalfunction:LimitOrderManager._collect()` and pass through to the `_pool.collect()` as minimum amounts to be collected (see Uniswap pool.collect doc)

UUC-01 | Missing Error Messages

Category	Severity	Location	Status
Coding Style	Informational	UniswapUtils.sol: 59, 82, 93, 134, 135, 136, 137, 138	Resolved

Description

The **require** can be used to check for conditions and throw an exception if the condition is not met. It is better to provide a string message containing details about the error that will be passed back to the caller.

Recommendation

We advise adding error messages to the linked **require** statements.

Alleviation

[Kromatika Team]: With the fix commit: b67ce36a150adaaa992e6810cd4bb84272552e46, Kromatika team have added short error messages to all require statements

UUC-02 | Potential Oracle Manipulation

Category	Severity	Location	Status
Logical Issue	Major	UniswapUtils.sol: 63	Partially Resolved

Description

In the function `quoteKROM()`, `timeWeightedAverageTick` is used to calculate the amount of KROM token received in exchange given a tick and a token amount. However, the `timeWeightedAverageTick` is fetched using Uniswap V3 oracle given the pool address and the `TWAP_PERIOD` that is used to calculate the time-weighted average. To be noticed, the `TWAP_PERIOD` is in units of seconds.

```
23     uint32 public constant TWAP_PERIOD = 20;
```

By the current setting, the `TWAP_PERIOD` is a constant and it's only 20 seconds, which means the `timeWeightedAverageTick` is vulnerable to oracle manipulation. The attacker could manipulate the pool for a few seconds to further manipulate the `timeWeightedAverageTick` since the time weight is relatively focused in a short period of time which makes it be easily manipulated.

Recommendation

Consider increasing the value of `TWAP_PERIOD` to increase the quote resilience from potential oracle manipulation.

Alleviation

[Kromatika Team]: Fix introduced with commit: bc0f18e75c1d376f18f19dc3465197e37047c85a initializing the `TWAP_PERIOD` to 1800 seconds.

UUC-03 | Potential Price Manipulation

Category	Severity	Location	Status
Logical Issue	Major	UniswapUtils.sol: 35, 122	Resolved

Description

In the function `calculateLimitTicks()`, Uniswap V3 pool's spot price is used to calculate the liquidity range.

```
35          (uint160 sqrtRatioX96,, , , , , ) = _pool.slot0();
```

The spot price of Uniswap V3 pools can be manipulated by flash loan attacks to generate a liquidity range that favors the attacker.

In the function `_amountsForLiquidity()`, Uniswap V3 pool's spot price is also used to compute the token0 and token1 value for a given amount of liquidity.

```
122          (uint160 sqrtRatioX96, , , , , , ) = pool.slot0();
```

The spot price of Uniswap V3 pools can be manipulated by flash loan attacks to generate token values that favor the attacker.

Recommendation

Considering using time weighted average price to reduce the effects from the pool price manipulation.

Alleviation

[Kromatika Team]: Kromatika team is acknowledging the recommendation and would like to describe the solution. In the solution, Kromatika is implementing buy / sell limit orders by adding a single-side liquidity on UniswapV3.Note that the function `calculateLimitTicks()` is getting an input parameter:_sqrtPriceX96thatdefines the user desired liquidity range for adding liquidity, that is different than the pool spot price:

```
(uint160 sqrtRatioX96,, , , , , ) = _pool.slot0();
```

The smart contracts are checking that the_sqrtPriceX96input parameter is greater for sell orders (or lower for buy orders) than the pool pricesqrtRatioX96.If that's not the case, including the case when the pool price has been changed / manipulated, the transaction would fail and it needs to fail, because UniswapV3 would not have accepted single-side liquidity with that pool price anyway. Using time-weighted average is not possible, since the check needs to be done against the current pool price in order for the single-side liquidity to be accepted by UniswapV3.

Update (Jan 31th 2022): Kromatika team with commit: f694074d5148a79938097b35111dbba965b545d4 has added a slippage (price manipulation) check introducing a minimum liquidity input parameters, based on a similar UniswapV3 check.

UUC-04 | Assumption Of Possitive Tick Spacing

Category	Severity	Location	Status
Logical Issue	Minor	UniswapUtils.sol: 34	Acknowledged

Description

In the function `calculateLimitTicks()`, the tick spacing of `IUniswapV3Pool _pool` is returned by `_pool.tickSpacing()` on line 34. The `tickSpacing` plays rather an essential role in the following calculations within the contract, for example, the `tickCeil` is considered the ceiling of the tick range by adding `tickSpacing` on the `tickFloor` assuming the `tickSpacing` is positive.

However, `tickSpacing` of certain pools may not be positive since the tick spacing could be defined as negative when constructing the pool, as its type suggests `int24`.

Recommendation

The audit team recommend adding certain validations on the `tickSpacing` of the `_pool`.

Alleviation

[Kromatika Finance]: Kromatika team is acknowledging the recommendation and would like to describe the solution. There is a _floor() function when calculating the `tickFloor`. The `tickSpacing` is used to find a liquidity range around `tickFloor` where a single-side liquidity is possible to be added on Uniswap. Regardless of whether `tickSpacing` is negative or positive, the smart contracts are checking both the ranges by adding or subtracting `tickSpacing` to determine which one is the possible single-side liquidity range, regardless of `tickSpacing` being positive or negative. [`tickFloor` - `tickSpacing`, `tickFloor`] - checking the one side of the liquidity range [`tickFloor`, `tickFloor` + `tickSpacing`] - checking the other side of the liquidity range.

UUC-05 | Redundant `import` Files

Category	Severity	Location	Status
Coding Style	Informational	UniswapUtils.sol: 16	Resolved

Description

The library `OracleLibrary` is imported twice and can be removed.

Alleviation

[Kromatika Team]: Cone cleanup has been performed removing duplicate imports https://github.com/Kromatika-Finance/limit-trade/blob/1ea8112beb6663dc69d841e846c95d118043939b/contracts/UniswapUtils.sol

UUC-06 | Integer Overflow/Underflow Risk

Category	Severity	Location	Status
Mathematical Operations	Informational	UniswapUtils.sol: 40, 43, 46	Resolved

Description

Using `+`/`-` in the method directly to calculate the value of the variable may overflow/underflow. `SafeMath` provides a method to verify overflow, and it is safer to use the method provided.

Recommendation

Consider using the `add()` and `sub()` function in `SafeMath` library for mathematical operations.

Alleviation

[Kromatika Team]: Team is acknowledging the recommendation and with the fix commit: cd8b10d2e6de1c3ba1dfb99570c0c865dec2b2f9, has introduced `SafeCast` and `SafeMath` libraries from OpenZepellin for safe mathematical and casting operations over uint256, uint128and uint32 types used in the contract.

UUC-07 | Proper Usage Of `require` And `assert` Functions

Category	Severity	Location	Status
Coding Style	Informational	UniswapUtils.sol: 93	Resolved

Description

The `assert()` function should only be used to test for internal errors, and to check invariants since the gas fee will not be returned. The `require()` function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts.

Recommendation

Consider using the `require()` function, along with a custom error message when the condition fails, instead of the `assert` function

Alleviation

[Kromatika Team]: With the fix commit: 66e112989600f8ff0b08dbb05de9c193d3ecf02f, Kromatika team has replaced `assert()` with `require()` function with a custom message.

Appendix

Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Mathematical Operations

Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.

Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS

AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING

MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

About

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.