Security Review Report
NM-0237 LayerAkira

NETHERMIND SECURITY

(June 28, 2024)

Contents

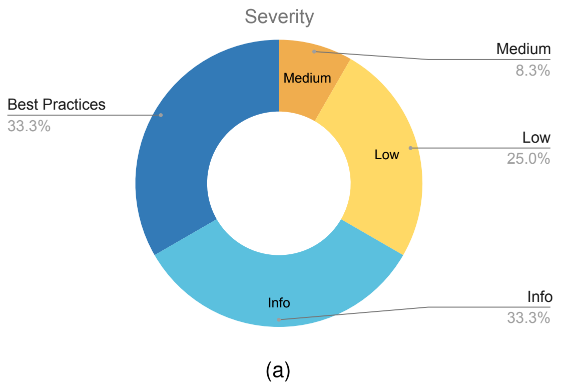
1 Executive Summary

This document outlines the security review conducted by Nethermind for the LayerAkira on-chain settlement layer. LayerAkira is a hybrid central limit orderbook protocol where users can submit signed orders to an off-chain orderbook which will match and execute trades through the LayerAkira Cairo contracts. Liquidity can be sourced from orders submitted by users, or routers, which are entities that allow funds to be sourced from other means outside of the LayerAkira ecosystem.

The audited code comprises 3210 lines of code. The `LayerAkira` team has provided comprehensive documentation explaining the behavior of protocol and the reviewed smart contracts. Aside from the provided documentation, the `LayerAkira` and `Nethermind Security` teams have communicated to clarify any remaining questions about the expected behavior of the protocol.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract. Along with this document, we report 12 points of attention, where one is classified as `Medium`, three are classified as `Low`, four are classified as `Info` and four are classified as `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 describes the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



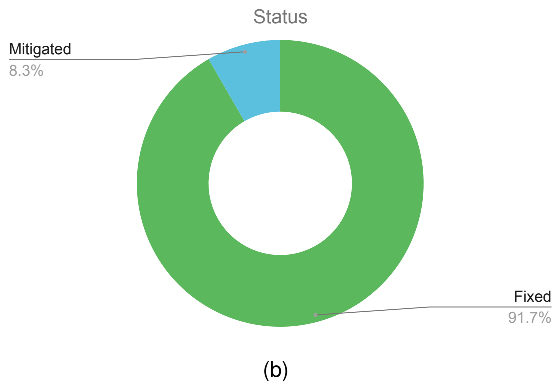


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (1), **Low** (3), **Undetermined** (0), **Informational** (4), **Best Practices** (4).
Distribution of status: Fixed (11), **Acknowledged** (0), **Mitigated** (1), **Unresolved** (0), **Partially Fixed** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	June 21, 2024
Response from Client	June 21, 2024
Final Report	June 28, 2024
Repository	LayerAkira/kurosawa_akira
Commit (Audit)	`f1cb9acbd363e16e36095ee132770ecdb726a885`
Commit (Final)	`94e206a70595e10428486449bca9b4432d9c5d16`
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/ILayerAkira.cairo	126	16	12.7%	49	191
2	src/FundsTraits.cairo	16	11	68.8%	4	31
3	src/lib.cairo	15	0	0.0%	0	15
4	src/WithdrawComponent.cairo	188	21	11.2%	47	256
5	src/NonceComponent.cairo	139	2	1.4%	19	160
6	src/EcosystemTradeComponent.cairo	888	26	2.9%	70	984
7	src/ExchangeBalanceComponent.cairo	283	3	1.1%	30	316
8	src/SignerComponent.cairo	90	7	7.8%	13	110
9	src/DepositComponent.cairo	77	3	3.9%	12	92
10	src/RouterComponent.cairo	332	38	11.4%	45	415
11	src/LayerAkira.cairo	457	2	0.4%	42	501
12	src/Order.cairo	300	7	2.3%	31	338
13	src/signature.cairo	3	0	0.0%	0	3
14	src/utils.cairo	4	0	0.0%	0	4
15	src/utils/erc20.cairo	21	2	9.5%	1	24
16	src/utils/common.cairo	17	0	0.0%	2	19
17	src/utils/account.cairo	17	0	0.0%	1	18
18	src/utils/SlowModeLogic.cairo	6	0	0.0%	1	7
19	src/signature/V0OffchainMessage.cairo	55	3	5.5%	6	64
20	src/signature/AkiraV0OffchainMessage.cairo	158	12	7.6%	16	186
21	src/signature/IOffchainMessage.cairo	18	0	0.0%	4	22
	Total	**3210**	**153**	**4.8%**	**393**	**3756**

3 Summary of Issues

	Finding	Severity	Update
1	User may not be able to apply on-chain withdrawals	Medium	Fixed
2	Receiver address for withdrawals is never checked	Low	Fixed
3	Router punishment can be abused to extract unclaimed router fees	Low	Fixed
4	fee_recipient address can be set to zero address	Low	Fixed
5	Missing validation could lead to deny new exchange version	Info	Fixed
6	Good After Time (GAT) orders can be filled before their valid date	Info	Mitigated
7	Tautology in the `_transfer` function	Info	Fixed
8	The `_do_part_external_taker_validate` function validates taker order's nonce with itself	Info	Fixed
9	Function/variable/comment corrections	Best Practices	Fixed
10	Inconsistency in error message	Best Practices	Fixed
11	Redundant checking in the function do_maker_checks	Best Practices	Fixed
12	`bind_to_signer` does not check signer address	Best Practices	Fixed

4 System Overview

LayerAkira Contract: Fig. 2 presents the component diagram for the LayerAkira on-chain settlement layer. The LayerAkira block (in yellow) represents the contract LayerAkira where all other components (in blue color) are imported. Admin only functionality such as the managment of versions, routers, whitelisted invokers, and the fee recipient is handled here. All other relevant functions from the components illustrated on the diagram are publicly exposed to whitelisted invokers and users through this contract.

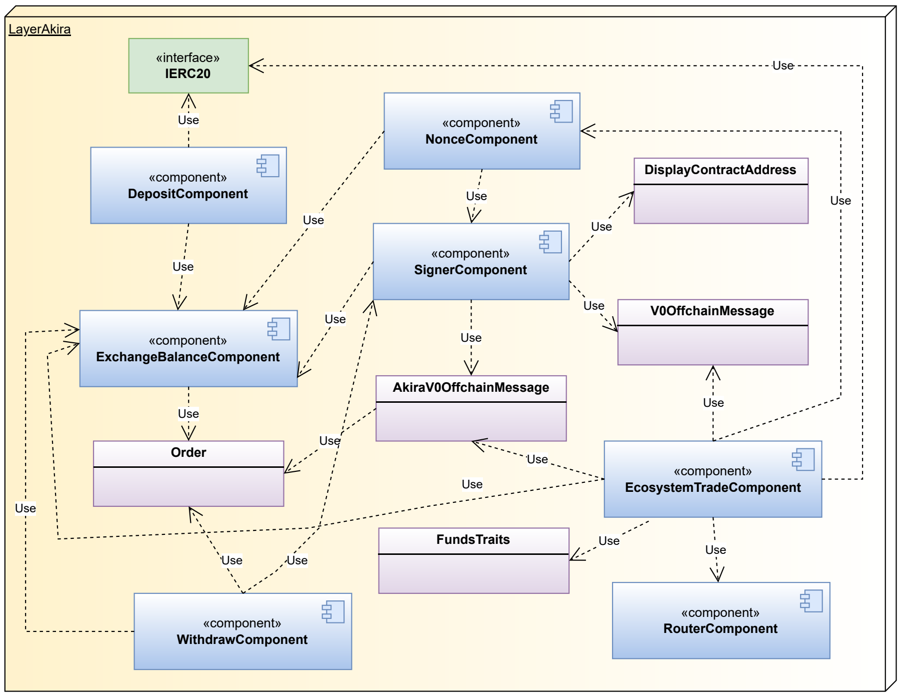

LayerAkira
«interface» IERC20
«component» NonceComponent
«component» DepositComponent
DisplayContractAddress
«component» SignerComponent
«component» ExchangeBalanceComponent
V0OffchainMessage
AkiraV0OffchainMessage
Order
«component» EcosystemTradeComponent
FundsTraits
«component» WithdrawComponent
«component» RouterComponent
Use

Fig. 2: Component diagram of the audited contracts

4.1 LayerAkira Components

Functions and Structs: There are several `cairo` files that contain structs and functions used by components. For example, the `Order` consists of several structs to organize data related to orders and functions to validate taker and maker orders.

DepositComponent: Allows users to directly deposit to the contract, where their funds can be used to create and sign orders to be submitted to the orderbook. This component relies on the `ExchangeBalanceComponent` to update balances upon a successful deposit.

EcosystemTradeComponent: Contains the majority of logic related to validation and settlement of matched orders. Two order match functions `apply_ecosystem_trades` and `apply_single_taker` are exposed by the `LayerAkira` contract allowing whitelisted invokers to execute orders. The punishment mechanism for routers which sign trades that are unexecutable is also handled within this component.

ExchangeBalanceComponent: Tracks user deposits and (partial) router balances as deposits and withdrawals occur. Additional functions exist to support gas payments to compensate for whitelisted invoker execution costs as well as balance adjustments after orders have been matched and executed.

NonceComponent: Handles nonces for users, with an public function allowing users to increase their nonce. These nonces are part of the order data, and are validated during execution to ensure that the nonce is not lower than the current. This allows users to invalidate all their existing orders if necessary.

RouterComponent: Logic for management of routers, which connects LayerAkira to external sources of liquidity. It contains functions for router registration, deposits, withdrawals and signer binding. Routers can also be de-registered with a delay, where they must first request and then apply.

SignerComponent: Contains logic for verifying that signatures are valid, as well as allowing traders to bind a signer address to their own address, meaning that signatures can be verified with a different address from the one that recieves funds.

WithdrawComponent: Mechanism for allowing users to withdraw funds in two different ways. The first approach is to have a whitelisted invoker execute the withdrawal. This will occur immediately and funds are sent to the recipient with no time delay. A user is also able to initiate a withdrawal of their funds at any time, with a certain delay. This delay gives the off-chain orderbook a grace period to react to any existing orders that may rely on the liquidity about to be removed.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Medium] User may not be able to apply on-chain withdrawals

File(s): `src/WithdrawComponent.cairo`

Description: The user can apply off-chain and on-chain withdrawals. When users call the `request_onchain_withdraw` function, it creates a pending withdrawal request for a particular token. Later, they need to invoke the `apply_onchain_withdraw` function to complete the ongoing request. On the other hand, they can withdraw by validating their signature. This issue emerges according to the order of calls, for example:

1. Alice calls `request_onchain_withdraw` to request an on-chain withdrawal of 100 A. Then, the function creates a pending request in `pending_reqs` for token A.

```
let key = (withdraw.token, withdraw.maker);
self.pending_reqs.write(key, (SlowModeDelay {block:get_block_number(), ts: get_block_timestamp()}, withdraw));
```

2. Alice decides not to wait the minimum period to apply the on-chain withdrawal through the `apply_onchain_withdraw` and executes the off-chain withdrawal of 100 tokens A by validating her signature.

3. Later, she wants to execute an on-chain withdrawal of 50 for A. She needs to create a new request because the pending one is with 100 tokens A. The `request_onchain_withdraw` function checks whether the previous pending request is completed, thus reverting the transaction.

```
fn request_onchain_withdraw(ref self: ComponentState<TContractState>, withdraw: Withdraw) {
    // ...
    let (pending_ts, w_prev): (SlowModeDelay, Withdraw)  = self.pending_reqs.read(key);
    let w_hash = withdraw.get_message_hash(withdraw.maker);

    assert!(w_prev != withdraw, "ALREADY_REQUESTED: withdraw for this token already requested");
    // @audit it will revert when checking if the request is completed.
    //        The hash for w_prev can be different from withdraw
    //        because we have several attributes in the
    //        Withdraw struct. Particularly, the values in the gas_fee
    //        struct tend to change
    assert!(w_prev.amount == 0 || self.completed_reqs.read(w_prev.get_message_hash(w_prev.maker)),
     ↪  "NOT_YET_COMPLETED_PREV: previous withdraw has not been completed yet");
   // ...
}
```

Then, Alice is forced to withdraw 100 tokens A (that she has already withdrawn) if she wants to continue to use the on-chain withdraw method. Otherwise, she will be able to **execute only off-chain withdrawals**. We can extend this example to **millions of tokens A where the user may not be able/interested to complete the pending request**.

Recommendation(s): Consider applying the same mechanism used for requesting and applying onchain unregister routers. For example:

– Set block and ts to zero when user calls in the `apply_withdraw` function: ;

```
fn apply_withdraw(ref self: ComponentState<TContractState>, signed_withdraw: SignedWithdraw, gas_price:u256,
 ↪  cur_gas_per_action:u32) {
//...
+let key = (signed_withdraw.withdraw.token, signed_withdraw.withdraw.maker);
-let (delay, w_req):(SlowModeDelay, Withdraw) = self.pending_reqs.read((signed_withdraw.withdraw.token,
 ↪  signed_withdraw.withdraw.maker));
+let (delay, w_req):(SlowModeDelay, Withdraw) = self.pending_reqs.read(key);

assert!(!self.completed_reqs.read(hash), "ALREADY_COMPLETED: withdraw (hash = {})", hash);

// @audit clear the pending_reqs
+self.pending_reqs.write(key, SlowModeDelay{block:0,ts:0});
// ...
}
```

– Apply a checking in the `request_onchain_withdraw` function to ensure that there is no ongoing pending request for withdrawal: ;

```
let (pending_ts, w_prev): (SlowModeDelay, Withdraw)  = self.pending_reqs.read(key);`
+assert!(pending_ts.block == 0, "ONCHAIN_WITHDRAW_ALREADY_REQUESTED: ...");
```

Status: Fixed

Update from the client: In this circumstance, the user may fix this issue by depositing funds back and fixing the pending withdrawal. As the Exchange does not charge a withdrawal fee, the User will only be losing out on gas in this scenario.

For a better user experience, the logic for apply_withdraw has been adjusted to invalidate any pending withdrawal if the user applies a withdrawal that is different from the scheduled one. This change will also apply if user wishes to invalidate their pending withdrawal.

See pull request #25 in commit: `94e206a70595e10428486449bca9b4432d9c5d16`

6.2 [Low] Receiver address for withdrawals is never checked

File(s): `src/WithdrawComponent.cairo`

Description: The on-chain withdrawal relies on pending requests that the user needs to create before withdrawing. However, the `request_onchain_withdraw` function does not verify the `receiver` address where the tokens will be transferred to.

```
fn request_onchain_withdraw(ref self: ComponentState<TContractState>, withdraw: Withdraw) {

    // @audit check if withdraw.receiver is not address zero. This validation is not applied in this function.
    assert!(get_caller_address() == withdraw.maker, "WRONG_MAKER: withdraw maker ({}) should be equal caller ({})",
    ↪  withdraw.maker, get_caller_address());
    assert!(withdraw.amount > 0, "WITHDRAW_CANT_BE_ZERO");
    // ...
    assert!(w_prev != withdraw, "ALREADY_REQUESTED: withdraw for this token already requested");
    assert!(w_prev.amount == 0 || self.completed_reqs.read(w_prev.get_message_hash(w_prev.maker)),
    ↪  "NOT_YET_COMPLETED_PREV: previous withdraw has not been completed yet");

    assert!(!self.completed_reqs.read(w_hash), "ALREADY_COMPLETED: requested withdraw has already been completed");
    // ...
}
```

When the caller invokes `apply_onchain_withdraw`, the function gets the pending request created for the particular token by `request_onchain_withdraw`.

```
fn apply_onchain_withdraw(ref self: ComponentState<TContractState>, token:ContractAddress, key:felt252) {
     let caller = get_caller_address();
    let (delay, w_req): (SlowModeDelay,Withdraw) = self.pending_reqs.read((token, caller));
    // ...
    self._transfer(w_req, key, w_req.amount, 0, true);
}
```

Once the user requests an on-chain withdraw, the `receiver` can not be changed and the `apply_onchain_withdraw` function reverts since the function `ERC20.transfer` reverts when `receiver` is zero address.

```
fn _transfer(ref self: ComponentState<TContractState>, w_req:Withdraw, w_hash:felt252, tfer_amount:u256,
↪  gas_price:u256, direct:bool) {
    // ...
    erc20.transfer(w_req.receiver, tfer_amount);
    // ...
}
```

Recommendation(s): Validate the `receiver` address in `request_onchain_withdraw`.

Status: Fixed

Update from the client: Added suggested asserts in pull request #25 in commit `94e206a70595e10428486449bca9b4432d9c5d16`

6.3 [Low] Router punishment can be abused to extract unclaimed router fees

File(s): src/EcosystemTradeComponent.cairo

Description: When an order is matched that uses external funds, a router must sign the order, and it should not fail during execution otherwise the router will be punished. The punishment is a percentage of the charged fee that the order would have cost to execute multiplied by two, and then split between the protocol fee recipient address and the maker order address.

```
fn punish_router_simple(...) {
    //...
    // Punishment is a percentage of the gas cost
    let charged_fee = gas_fee.gas_per_action.into()
        * gas_px
        * router.get_punishment_factor_bips().into()
        / 10000;
    if charged_fee == 0 { return; }
    // Router loses 2x the charged fee
    // One half goes to protocol, other half goes to maker order address
    router.burn(router_addr, native_base_token, 2 * charged_fee);
    balancer.mint(balancer.fee_recipient.read(), charged_fee, native_base_token);
    balancer.mint(maker, charged_fee, native_base_token);
    // ...
}
```

As part of the calculation for `charged_fee` the gas price is multiplied by the gas per action. The `gas_px` argument is provided by the caller which is a trusted whitelisted invoker address, however the `gas_per_action` is provided from the `gas_fee` field from the taker order's data. This allows a taker to arbitrarily control the punishment fee that a router would incur if execution of an order failed.

To avoid punishments, the router is expected to verify that the taker's order is valid by simulating it's execution. A taker can construct a custom smart wallet implementation that will return a valid signature on simulation, but fail during execution. This can be done by inspecting the `get_tx_info().unbox().version` field and checking for the "simulate" bit at position 128. Another approach could be to simply use a regular contract instead, where the function `is_valid_signature` can access state and arbitrarily return whether a signature is valid or not.

Since the punishment fee imposed on the router is split between the protocol and the maker order, an attacker can construct a maker order and a taker order that will immediately match with each other. The taker order would have a `gas_fee.gas_per_action` which is set to a value such that after it is multiplied with the current `gas_px`, adjusted by the punishment factor and multiplied by two, is equal (or very close to) the balance of the router address. When the punishment is applied, the attacker will receive half of the unclaimed fees belonging to the router since the maker order also belongs to them.

This finding relies on offchain components to behave a certain way to be feasible. The router must accept user order data rather than generating data and submitting it to the user to sign. The router also should miss validity checks on the gas steps field. Validations on the expected execution cost would need to check for profit after execution, with no upper bounds (if user offers excessive gas payment it would accept). In order to profit from this attack, an attacker must match their own taker and maker order, which may be challenging depending on factors such as spread and volatility. If the attacker's taker order matches with another user's maker order, they will receive the stolen funds instead.

In summary, an attacker can construct self-matching maker/taker orders and manipulate their `gas_per_action` field with a specific value that will result in the punishment fee being equal to a router's unclaimed fee balance. The attacker can guarantee that punishment will occur by using a custom wallet implementation, and when the punishment is applied they will receive half of the router's unclaimed fees.

Recommendation(s): Consider adjusting the `punish_router_simple` function to use the `cur_gas_per_action` argument provided by the whitelisted invoker in`apply_single_taker`, instead of using the untrusted `gas_per_action` provided by the taker order. This will prevent the punishment fee from being controlled by the taker order.

Status: Fixed

Update from the client:

This issue relates only to the unclaimed portion of fees the Router earns through orders routed to LayerAkira.

The likelihood of these fees getting drained is highly unlikely and would only arise in a scenario where the maker, taker and exchange all act negligently / maliciously. This is because:

- LayerAkira will be applying the rollups on the exchange and will require users to use a specific value for gas_per_action since we know the upper bound of what the user would spend ;
- If more trades can fit into a rollup, then the Exchagne will adjust the steps per user, making it cheaper per user. The point here is that this is a constant field;
- The Taker cannot perform this attack as the Exchange will be doing checks and marking these types of orders as invalid ;
- The risk is further mitigated when the Router performs basic checks on the Taker, which is what the Router receives its portion of the Router Rewards for (and therefore is in its best interest to do);

The suggested asserts have also been implemented to make this scenario even more unlikely in pull request #25 commit `de4853abe8e3ff9a6d75ca3b38db20af757b9f6d`.

6.4 [Low] fee_recipient address can be set to zero address

File(s): src/LayerAkira

Description: The owner calls update_fee_recipient to update the fee recipient address. The fee_recipient receives tokens from the users when they pay fees. However, the owner could set a zero address by mistake since the function does not check the new_fee_recipient value. The owner can easily fix this issue, but the protocol may lose tokens until the problem is detected.

```
#[external(v0)]
fn update_fee_recipient(ref self: ContractState, new_fee_recipient: ContractAddress) {
    assert!(self.owner.read() == get_caller_address(), "Access denied: update_fee_recipient is only for the owner's
     ↪  use");
    // @audit fee_recipient should always be different from zero
    //        since fees are transferred to this address
    self.balancer_s.fee_recipient.write(new_fee_recipient);
    self.emit(FeeRecipientUpdate{new_fee_recipient});
}
```

Recommendation(s): Consider adding a check if new_fee_recipient is not zero address.

Status: Fixed

Update from the client: We have addressed the recommendation in pull request #25 in commit 94e206a70595e10428486449bca9b4432d9c5d16

6.5 [Info] Missing validation could lead to deny new exchange version

File(s): src/LayerAkira.cairo

Description: Before filling orders, generic_taker_check(...) validates if maker and taker are using the same exchange version and to update the contract's version, only versions higher than the current one are allowed:

```
    fn update_exchange_version(... , new_version:u16) {
        // @audit Condition to enforce version incrementation
        assert!(new_version > self.exchange_version.read(), "Exchange version can only increase");
        // ...
    }
```

The problem is that in case of malicious or unintentional upgrade to u16 maximum value the contract won't admit new exchange versions.

Recommendation(s): Add a validation in update_exchange_version to restrict the difference between the current and new version numbers.

Status: Fixed

Update from the client: Addressed in pull request #25 commit f4ee3230a2c99b22398de5027bcd4adcad804fb4. The field has been removed as it no longer has any business logic behind it. The signature logic was adjusted to factor in the account contract address. This will avoid replicating fills of orders that users submit for the original exchange. Users can be assured that the orders they sign are only valid on the particular version of the exchange.

6.6 [Info] Good After Time (GAT) orders can be filled before their valid date

File(s): `src/Order.cairo`

Description: A good-after-time (GAT) order is a trading instruction which can be combined with market orders, limit orders and other order types. A GAT order instructs to wait until a predetermined time or date has transpired before order become valid.

To create an order, market participants specify creation and duration timestamps that define the time period in which the order is considered valid and can be filled by takers. The timestamps are validated in `do_maker_checks(...)` and `generic_taker_check(...)` functions as part of the order fulfilment process. The problem is that these functions lack checks to determine if the current timestamp is greater than the `created_at` attribute of the order.

```
fn do_maker_checks(...) {
    // ...
    // @audit This condition just validates the end of valid time.
    assert!(get_block_timestamp() < maker_order.constraints.created_at.into() +
    ↪  maker_order.constraints.duration_valid.into(), "Maker order expire {}",
    ↪  maker_order.constraints.duration_valid);
    // ...
}
```

```
fn generic_taker_check(...) {
    // ...
    // @audit This condition just validates the end of valid time.
    assert!(get_block_timestamp() <  taker_order.constraints.created_at.into() +
    ↪  taker_order.constraints.duration_valid.into(), "Taker order expire {}",
    ↪  taker_order.constraints.duration_valid);
}
```

Recommendation(s): Before filling the order, consider validating that the timestamp is greater or equal to the order creation timestamp.

Status: Mitigated

Update from the client: This issue is addressed with the Exchange's off-chain validation logic. When orders are sent to the Exchange, orders created too early are marked as stale and therefore not processed. This check was moved off-chain so that it does not present friction when the Exchange is performing rollups.

6.7 [Info] Tautology in the `_transfer` function

File(s): `src/WithdrawComponent.cairo`

Description: The function `_transfer` is called during withdrawals. This function receives the boolean parameter `direct`.

```
fn _transfer(ref self: ComponentState<TContractState>,
             w_req:Withdraw,
             w_hash:felt252,
             tfer_amount:u256,
             gas_price:u256,
             direct:bool) {
             // ...
}
```

The `apply_withdraw` function calls `_transfer`, passing the value of the comparison of `w_req == signed_withdraw.withdraw`. As presented below, it invokes `_transfer`, `w_req` always equal to `signed_withdraw.withdraw`.

```
fn apply_withdraw(ref self: ComponentState<TContractState>, signed_withdraw: SignedWithdraw, gas_price:u256,
↪  cur_gas_per_action:u32) {
    let hash = signed_withdraw.withdraw.get_message_hash(signed_withdraw.withdraw.maker);
    let (delay, w_req):(SlowModeDelay, Withdraw) = self.pending_reqs.read((signed_withdraw.withdraw.token,
    ↪  signed_withdraw.withdraw.maker));
    // ...
    // @audit w_req is now equal to signed_withdraw.withdraw
    let w_req = signed_withdraw.withdraw;
    // ...
    // @audit-info Tautology: the result of w_req == signed_withdraw.withdraw is always true
    self._transfer(w_req, hash, tfer_amount, gas_price, w_req == signed_withdraw.withdraw);
}
```

Similarly, the `apply_onchain_withdraw` function always passes true to `_transfer`.

```
fn apply_onchain_withdraw(ref self: ComponentState<TContractState>, token:ContractAddress, key:felt252) {
    // ...
    self._transfer(w_req, key, w_req.amount, 0, true);
}
```

Recommendation(s): Consider removing this parameter or reevaluating the goal for the `direct` parameter.

Status: Fixed

Update from the client: The `apply_onchain_withdraw` function always passes true to `_transfer` because this flag in the event indicates the origin of the withdrawal, whether it is generated off-chain or on-chain. Since apply_onchain_withdraw is on-chain, the flag is always set to true.

In contrast, when the exchange performs a rollup with user withdrawals, the exchange need to distinguish whether it is the finalization of an on-chain withdrawal or purely an off-chain one. This has now been addressed by performing this comparison before reassigning `w_req` to `signed_withdraw.withdraw`.

Addressed in pull request #25 in commit `94e206a70595e10428486449bca9b4432d9c5d16`

6.8 [Info] The `_do_part_external_taker_validate` function validates taker order's nonce with itself

File(s): `src/EcosystemTradeComponent.cairo`

Description: The function `apply_single_taker` invokes `_do_part_external_taker_validate` when the taker order defines that the source of the funds to execute the trade are not from the balance of the Exchange, i.e., it is flagged `external_funds=true`. As presented below, `_do_part_external_taker_validate` calls `generic_taker_check` function to validate data related to taker orders.

```
        fn _do_part_external_taker_validate(self:@ComponentState<TContractState>,
            signed_taker_order:SignedOrder, swaps:u16, version: u16, fee_recipient:ContractAddress) ->
             ↪  (Order,felt252,OrderTradeInfo, u256) {
            // ...
            // @audit the taker_order's nonce is passed instead the one in NonceComponent.get_nonce(taker_order.maker)
            super::generic_taker_check(taker_order, taker_fill_info, taker_order.constraints.nonce, swaps, taker_hash,
             ↪  version, fee_recipient);
           // ...
        }
```

The `generic_taker_check` function receives several information associated to the taker order, such as the `taker_order` and the `nonce` of the current order (instead get from `NonceComponent.get_nonce(taker_order.maker)`. Then, the function checks whether the taker orders' nonce is greater than or equal to `nonce`. This checking will be always true, since the `nonce` parameter represents the nonce assigned in the taker order.

```
fn generic_taker_check(taker_order:Order, taker_fill_info:OrderTradeInfo, nonce:u32, swaps:u16,
 ↪  taker_order_hash:felt252, version:u16, fee_recipient:ContractAddress) {
    // ...
    assert!(taker_order.constraints.nonce >= nonce, "OLD_TAKER_NONCE");
    // ...
}
```

This error does not trigger any vulnerability because the taker's nonce is correctly implemented in the `_prepare_router_taker` function that is called later by `apply_single_taker`. However, `_do_part_external_taker_validate` may be reused somewhere in future changes and it could add issues.

Recommendation(s): Consider applying the following change:

```
-super::generic_taker_check(taker_order, taker_fill_info, taker_order.constraints.nonce, swaps, taker_hash, version,
 ↪  fee_recipient);
+super::generic_taker_check(taker_order, taker_fill_info, contract.get_nonce(taker_order.maker), swaps, taker_hash,
 ↪  version, fee_recipient);
```

Status: Fixed

Update from the client: Addressed in pull request #25 in commit `de4853abe8e3ff9a6d75ca3b38db20af757b9f6d`.

6.9 [Best Practice] Inconsistency in error message

File(s): src/LayerAkira.cairo

Description: The owner invokes `update_exchange_version` to update the exchange version. When the caller is not the owner, it triggers an error message of access denied to the `update_exchange_invorkers` function instead of `update_exchange_version`.

```
#[external(v0)]
fn update_exchange_version(ref self: ContractState, new_version:u16) {
  // @audit the error message is referring to update_exchange_invokers function
  assert!(self.owner.read() == get_caller_address(), "Access denied: update_exchange_invokers is only for the owner's
  ↪  use");
  assert!(new_version > self.exchange_version.read(), "Exchange version can only increase");
  self.exchange_version.write(new_version);
  self.emit(VersionUpdate{new_version});
}
```

Recommendation(s): Consider adapting the error message to the proper function.

Status: Fixed

Update from the client: Adapted the error message to the properly function in pull request #25 in commit `94e206a70595e10428486449bca9b4432d9c5d16`

6.10 [Best Practice] `bind_to_signer` does not check signer address

File(s): src/SignerComponent.cairo

Description: The caller invokes `bind_to_signer` function to set a signer. The function checks if the signer is already set, updates the storage, and emits an event with the trader account and signer. However, it does not check whether `signer` is different from zero address.

```
fn bind_to_signer(ref self: ComponentState<TContractState>, signer: ContractAddress) {
    let caller = get_caller_address();
    // @audit check if signer is not zero address
    assert!(self.trader_to_signer.read(caller) == 0.try_into().unwrap(), "ALREADY BINDED: signer = {}",
    ↪  self.trader_to_signer.read(caller));
    self.trader_to_signer.write(caller, signer);
    self.emit(NewBinding { trading_account: caller, signer: signer });
}
```

Recommendation(s): Consider checking if `signer` is not a zero address.

Status: Fixed

Update from the client: Addressed recommendation in pull request #25 in commit `94e206a70595e10428486449bca9b4432d9c5d16`

```
self.trader_to_signer.read(caller) == 0.try_into().unwrap()
```

6.11 [Best Practices] Function/variable/comment corrections

File(s): src/*

Description: The following is a list of functions, variables and comments that can be corrected:

`EcosystemTradeComponent.cairo`

- Function `can_tranfer` -> `can_transfer` ;
- Function `trasfer_in` -> `transfer_in` ;
- Function `trasfer_back` -> `transfer_back` ;
- Assert message in `apply_fixed_fees`, `MOSMATCH` -> `MISMATCH` ;

Recommendation(s): Consider implementing the corrections mentioned above.

Status: Fixed

Update from the client: Addressed in pull request #25 in commit `de4853abe8e3ff9a6d75ca3b38db20af757b9f6d`

6.12 [Best Practices] Redundant checking in the function do_maker_checks

File(s): `src/Order.cairo`

Description: The function `do_maker_checks` applies redundant checking for `full_fill_only`. It asserts that the `maker_order` is not `full_fill_only`. However, when the flag `post_only = true`, the function applies again the same checking.

```
fn do_maker_checks(maker_order:Order, maker_fill_info:OrderTradeInfo, nonce:u32,fee_recipient:ContractAddress)-> (u256,
↪ u256) {
    // ...
    assert!(!maker_order.flags.full_fill_only, "WRONG_MAKER_FLAG: maker_order can't be full_fill_only");
    assert!(get_block_timestamp() < maker_order.constraints.created_at.into() +
    ↪ maker_order.constraints.duration_valid.into(), "Maker order expire {}",
    ↪ maker_order.constraints.duration_valid);
    if maker_order.flags.post_only {
        // @audit Redundant checking for !maker_order.flags.full_fill_only
        //        at this point, maker_order.flags.full_fill_only == false (it was checked above in the function)
        assert!(!maker_order.flags.best_level_only && !maker_order.flags.full_fill_only, "WRONG_MAKER_FLAGS");
    }
    // ...
}
```

Recommendation(s): Consider removing the redundant checking.

Status: Fixed

Update from the client: Addressed in pull request #25 in commit `de4853abe8e3ff9a6d75ca3b38db20af757b9f6d`.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the LayerAkira Protocol documentation

The documentation for LayerAkira is available on LayerAkira gitbook. It consists of sections with:

- General overview of the system.
- Technical Overview to specify fee types, exchange flow, trading pipeline and accounts, ecosystem book, router book, and routers.
- Trade section explaining details about the fees, order, deposit, and withdrawal.
- Several examples to illustrate the expected behavior for the main features.

The information in the documentation is concise and technical, with well-written explanations. **Code comments have a reasonable quality**, with descriptions for the main structs. **During the audit, the LayerAkira Team had effective communication**. They were available for meetings to answer questions, explain further, and discuss the detected issues. In addition, they were consistently accessible, ensuring that our team could seek and receive timely clarifications and support as needed.

8 Test Suite Evaluation

8.1 Contracts Compilation

```
$ scarb --version; snforge --version
scarb 2.6.3 (e6f921dfd 2024-03-13)
cairo: 2.6.3 (https://crates.io/crates/cairo-lang-compiler/2.6.3)
sierra: 1.5.0
snforge 0.23.0

$ scarb build
    Compiling kurosawa_akira v0.1.0 (/home/dev/dev/nm/synnax/gauss/NM-0237/report/kurosawa_akira/Scarb.toml)
    warn: Unused variable. Consider ignoring by prefixing with `_`. (occurred four times, output omitted)
    Finished release target(s) in 10 seconds
```

8.2 Tests Output

```
$ snforge test

Collected 39 test(s) from kurosawa_akira package
Running 39 test(s) from src/

[PASS] tests_deposit_and_withdrawal_and_nonce::dd
[PASS] tests_router_trade::test_cant_execute_with_not_registered_router
[PASS] tests_ecosystem_trade::test_succ_match_single_sell_taker_trade_full
[PASS] tests_quote_qty_ecosystem_trade_01::test_double_qty_SELL_maker_05_double
[PASS] tests_quote_qty_ecosystem_trade_01::test_double_qty_BUY_maker_01_match_quote_qty
[PASS] tests_ecosystem_trade::test_succ_match_single_buy_taker_trade_full
[PASS] tests_quote_qty_ecosystem_trade_02::test_quote_qty_only_base
[PASS] tests_quote_qty_ecosystem_trade_02::test_quote_qty_only_quote
[PASS] tests_quote_qty_router_trade_02::test_quote_qty_both_02
[PASS] tests_deposit_and_withdrawal_and_nonce::test_increase_nonce
[PASS] tests_quote_qty_ecosystem_trade_01::test_double_qty_SELL_maker_02_match_quote_qty
[PASS] tests_deposit_and_withdrawal_and_nonce::test_withdraw_eth_direct_no_delayed_by_exchange
[PASS] tests_ecosystem_trade::test_succ_match_single_buy_taker_trade_full_router
[PASS] tests_ecosystem_trade::test_succ_match_single_sell_taker_trade_full_router
[PASS] tests_quote_qty_ecosystem_trade_01::test_double_qty_SELL_maker_01
[PASS] tests_ecosystem_trade::test_double_qty_SELL_maker_01_oracle_qty
[PASS] tests_deposit_and_withdrawal_and_nonce::test_eth_deposit
[PASS] tests_router_trade::test_execute_with_buy_taker_succ
[PASS] tests_quote_qty_ecosystem_trade_01::test_double_qty_SELL_maker_04_double
[PASS] tests_deposit_and_withdrawal_and_nonce::test_withdraw_eth_direct_immediate
[PASS] tests_deposit_and_withdrawal_and_nonce::test_withdraw_eth_indirect_twice
[PASS] tests_deposit_and_withdrawal_and_nonce::test_withdraw_eth_direct_delayed
[PASS] tests_router_trade::test_execute_with_buy_taker_succ_spent_side_fee_for_both_parties
[PASS] tests_deposit_and_withdrawal_and_nonce::test_withdraw_eth_indirect
[PASS] tests_quote_qty_router_trade_01::test_roter_trade_double_qty_semantic_SELL_maker_02_match_quote_qty
[PASS] tests_quote_qty_router_trade_01::test_roter_trade_double_qty_semantic_SELL_maker_03_match_base_qty
[PASS] tests_quote_qty_router_trade_02::test_quote_qty_only_quote
[PASS] tests_quote_qty_router_trade_01::test_roter_trade_double_qty_semantic_BUY_maker_01
[PASS] tests_quote_qty_ecosystem_trade_02::test_quote_qty_both_02
[PASS] tests_quote_qty_router_trade_01::test_roter_trade_double_qty_semantic_BUY_maker_02_match_quote_qty
[PASS] tests_quote_qty_router_trade_01::test_roter_trade_double_qty_semantic_BUY_maker_03_match_base_qty
[PASS] tests_quote_qty_router_trade_01::test_roter_trade_double_qty_semantic_SELL_maker_01
[PASS] tests_router_trade::test_punish_router
[PASS] tests_deposit_and_withdrawal_and_nonce::test_withdraw_eth_direct_delayed_cant_apply_twice
[PASS] tests_quote_qty_router_trade_02::test_quote_qty_both
[PASS] tests_quote_qty_ecosystem_trade_01::test_double_qty_SELL_maker_03_match_base_qty
[PASS] tests_quote_qty_router_trade_02::test_quote_qty_only_base
[PASS] tests_router_trade::test_execute_with_sell_taker_succ
[PASS] tests_quote_qty_ecosystem_trade_02::test_quote_qty_both

Tests: 39 passed, 0 failed, 0 skipped, 0 ignored, 0 filtered out
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer