Security Review Report
NM-0580 Hanji CLOB



NETHERMIND
SECURITY

(November 3, 2025)

Contents

1 Executive Summary **2**

2 Audited Files **3**

3 Summary of Issues **3**

4 System Overview **4**
4.1 Contract Interaction Diagram 4

5 Risk Rating Methodology **5**

6 Issues **6**
6.1 [High] Proxy orders use the owner's deposited funds instead of the proxy's funds 6
6.2 [High] `OnchainCLOB` markets with non-native token pairs can be halted 7
6.3 [Info] ERC20 Permit support 7
6.4 [Info] Order claiming allows token transfers and is open by default 8
6.5 [Info] Post-only orders can be griefed via front-running 8
6.6 [Info] Proxy orders refund dust from `msg.value` to the order owner 9
6.7 [Info] The function `getAccumulatedFees` returns a scaled value 10
6.8 [Best Practices] Function visibilities can be stricter 10
6.9 [Best Practices] Unnecessary/unused imports 11
6.10 [Best Practices] Unused errors 11

7 Documentation Evaluation **12**

8 Test Suite Evaluation **13**
8.1 Compilation Output 13
8.2 Tests Output 13

9 About Nethermind **14**

1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security for Hanji's `OnchainCLOB` and `Trie` contracts. Hanji is a fully onchain order book exchange, where all orders are placed, matched and managed onchain. Users are able to create market orders or limit orders, with a choice on whether the order should only be posted without executing, or to execute immediately. The `OnchainCLOB` contract provides functions for users to manage their orders and handles asset deposit withdrawals, while the `Trie` contract contains a radix trie implementation for storing and matching user's orders.

The audit comprises 2541 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools, and (c) creation of test cases.

Along this document, we report 11 points of attention, where one is classified as `Critical`, two are classified as `High`, one is classified as `Low`, and seven are classified as `Informational` or `Best Practices` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.

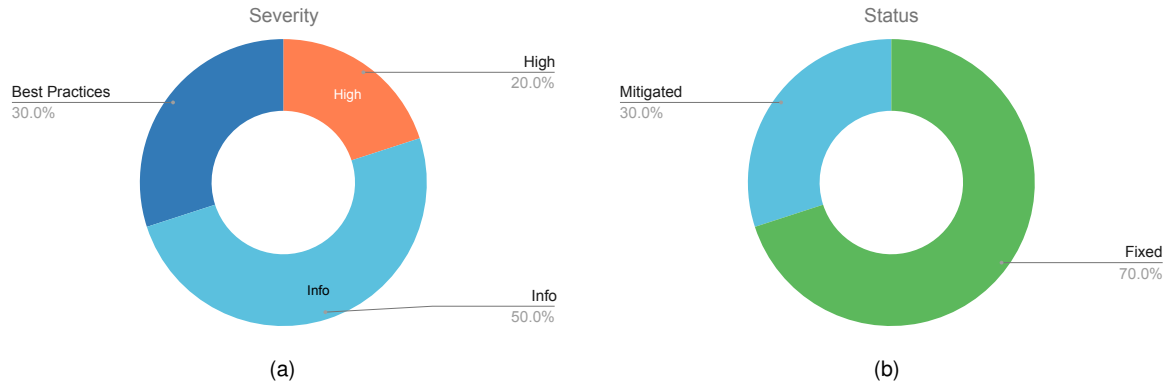


Fig. 1: Distribution of issues: Critical (0), **High** (2), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (5), **Best Practices** (3). **Distribution of status: Fixed** (7), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	August 28, 2025
Final Report	November 3, 2025
Initial Commit Hash	`4c8237da4ec356ad1a96b7308c1a8ae5f41b23a5`
Final Commit Hash	`512320f21c7fbd6eadbfb698a9b44a917e5ed96b`
Documentation Assessment	Medium
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/Errors.sol	36	3	8.3%	2	41
2	src/ITradeConsumer.sol	4	3	75.0%	2	9
3	src/IOnchainCLOBFactory.sol	20	3	15.0%	2	25
4	src/ITrie.sol	12	3	25.0%	2	17
5	src/IWatchDog.sol	5	3	60.0%	2	10
6	src/IWETH.sol	5	3	60.0%	2	10
7	src/WatchDog.sol	50	5	10.0%	19	74
8	src/Proxy.sol	5	3	60.0%	4	12
9	src/WatchDogFactory.sol	38	11	28.9%	11	60
10	src/OnchainCLOBFactory.sol	101	31	30.7%	18	150
11	src/ITrieFactory.sol	4	3	75.0%	2	9
12	src/Helper.sol	196	50	25.5%	31	277
13	src/Trie.sol	877	285	32.5%	229	1391
14	src/OnchainCLOB.sol	1023	260	25.4%	171	1454
15	src/IOnchainCLOB.sol	133	3	2.3%	19	155
16	src/FP24.sol	23	9	39.1%	10	42
17	src/TrieFactory.sol	9	8	88.9%	5	22
	Total	**2541**	**686**	**27.0%**	**531**	**3758**

3 Summary of Issues

	Finding	Severity	Update
1	Proxy orders use the owner's deposited funds instead of the proxy's funds	High	Fixed
2	`OnchainCLOB` markets with non-native token pairs can be halted	High	Fixed
3	ERC20 Permit support	Info	Acknowledged
4	Order claiming allows token transfers and is open by default	Info	Fixed
5	Post-only orders can be griefed via front-running	Info	Fixed
6	Proxy orders refund dust from `msg.value` to the order owner	Info	Acknowledged
7	The function `getAccumulatedFees` returns a scaled value	Info	Fixed
8	Function visibilities can be stricter	Best Practices	Fixed
9	Unnecessary/unused imports	Best Practices	Fixed
10	Unused errors	Best Practices	Fixed

4 System Overview

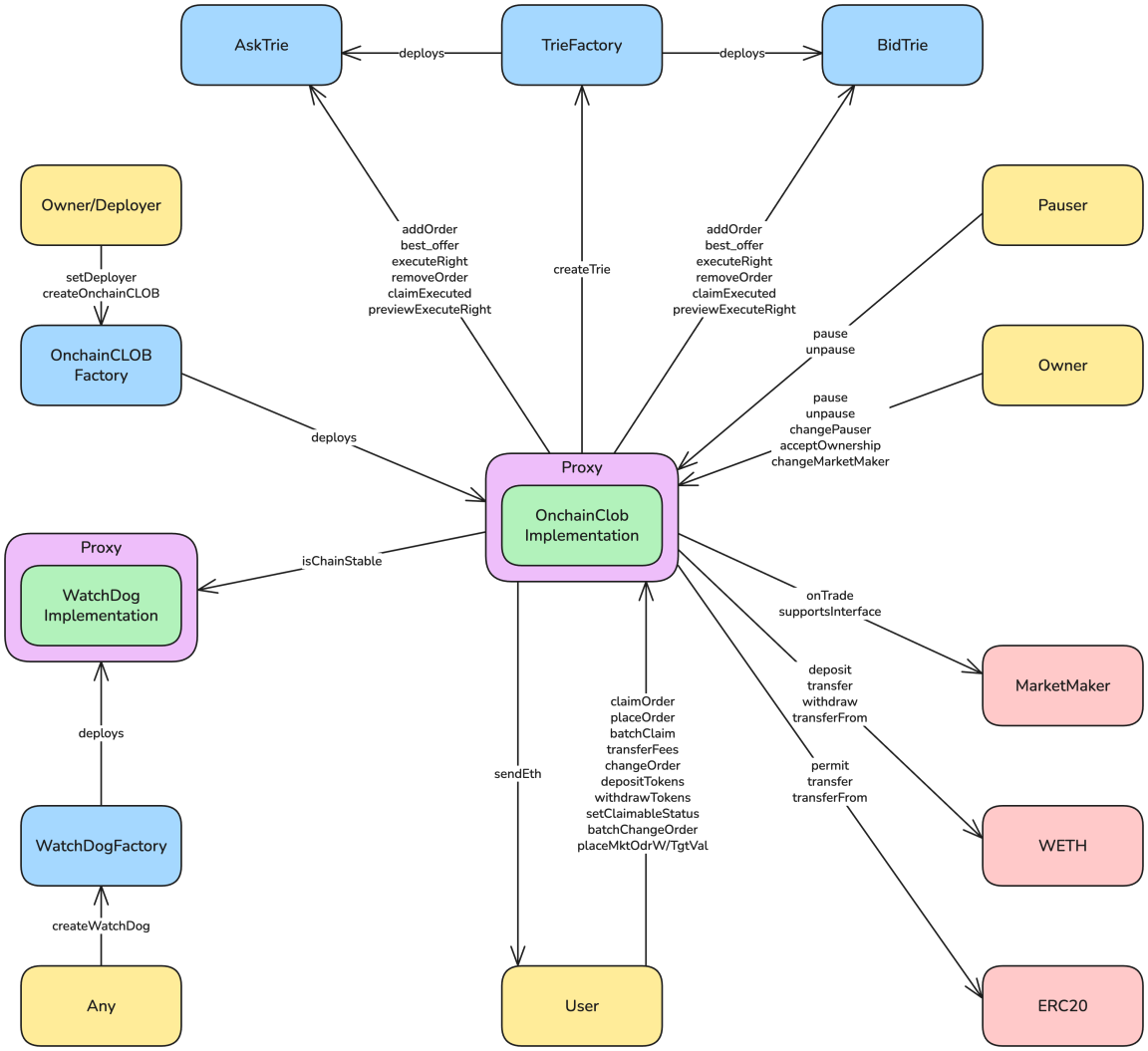
The Hanji onchain order book protocol primarily consists of three separate contract types: The `OnchainCLOB`, `Trie`, and the `Watchdog`. Some contracts also have factories associated with them, which are shown in the contract interaction diagram in the following subsection.

OnchainCLOB: The primary contract through which both users and permissioned roles interact. Users can place market or limit orders, with options to execute immediately or post only to the order book. This contract also manages token deposits, withdrawals, and overall asset management. To optimize gas costs for high-frequency traders, it includes functionality for storing idle capital directly within the contract, reducing repeated transfer overhead. Orders may be cancelled or claimed even when only partially filled, with the choice to withdraw the associated assets or leave them in the contract under internal accounting.

Trie: This contract is deployed twice, once for tracking bid orders and once for tracking ask orders. Its entry points are strictly restricted, permitting access only from the `OnchainCLOB` contract. Orders are stored efficiently in a radix trie and can be executed and claimed either fully or partially.

Watchdog: This contract functions as a safeguard against chain-level issues that block transaction processing. Each interaction with the watchdog updates a timestamp recording the last activity. If transactions cannot be processed for a sufficiently long period, the watchdog prevents the `OnchainCLOB` from placing new orders. This mechanism provides a grace period in which pending orders can be removed or adjusted, mitigating risks from potential market shifts during times when the chain temporarily prevents order modifications.

4.1 Contract Interaction Diagram



5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security 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 Security 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 [High] Proxy orders use the owner's deposited funds instead of the proxy's funds

File(s): src/OnchainCLOB.sol

Description: The `placeOrderByProxy(...)` function allows an approved proxy address to place orders on behalf of a user (`order_owner`). This functionality is intended for Request-for-Quote (RFQ) systems, where a market maker acts as the proxy to execute a trade for a user. The expected behavior is that the proxy (`msg.sender`) provides the funds for the order it initiates.

However, the underlying payment logic in the `_handleTokenTransfer(...)` function does not distinguish between a proxy order and a regular order. When sourcing funds, the function first depletes the `order_owner`'s internal token balance (`traderBalances`) before requesting any funds from the proxy.

The `placeOrderByProxy(...)` function calls the internal `_placeOrder(...)` function, which in turn calls `_handleTokenTransfer(...)` to manage the movement of assets.

```
function placeOrderByProxy(
    // ...
    address order_owner,
    // ...
) public payable /* ... */ {
    // ...
    // @audit Calls _placeOrder with the user as `order_owner` but msg.sender as the proxy.
    ExecutionResult memory result = _placeOrder(
        // ...
        order_owner
    );
    // ...
}
```

The core issue lies within `_handleTokenTransfer(...)`, where `client` is the `order_owner`. The logic first checks and uses the `client`'s deposited funds.

```
function _handleTokenTransfer(
    address client, // @audit This is the `order_owner`
    ...
) internal {
    // ...
    // processing token_x
    if (x_to_send < x_to_receive) {
        // ...
        uint128 clients_shares = traderBalances[client].token_x;
        // @audit It the `order_owner` has enough balance then funds are taken from their balance
        if (clients_shares >= shares_to_receive) {
            traderBalances[client].token_x -= shares_to_receive;
            // ...
        }
    }
    // ...
    if (actual_token_x_to_receive > 0) {
        _safeTransferFromWithBalanceCheck(
            token_x,
            msg.sender, // @audit If user's balance doesn't cover the full order, proxy pays the rest
            address(this),
            actual_token_x_to_receive
        );
    }
    // ...
}
```

Instead of the proxy covering the full cost of the order, the user's already existing funds in the OnchainCLOB contract are used first. As an example attack, this vulnerability can be exploited by a malicious user with proxy permissions who can use the `placeOrderByProxy(...)` function to create spread in the orderbook, and then create orders on behalf of the user that match with their own, moving back and forth within the created spread, to effectively arbitrage funds out of the user and into the malicious proxy account.

Recommendation(s): Consider modifying the token transfer logic to differentiate between orders placed directly by a user and those placed by a proxy.

Status: Fixed

Update from the client: Fixed in commit 32439ce7. The function `placeMarketOrderWithTargetValueByProxy` has also been implemented.

6.2 [High] `OnchainCLOB` markets with non-native token pairs can be halted

File(s): `src/OnchainCLOB.sol`

Description: The `OnchainCLOB` contract supports ERC20 token pairs where one of the tokens can be a wrapped native asset (e.g., Wrapped Ether). This is controlled by the storage variable `supports_native_eth`, which, if set to `true`, indicates that the market accepts the chain's native assets as payment. Internally, the native asset is converted into the wrapped ERC20 token during deposits and converted back into the native asset during withdrawals.

Token inflow and outflow for order execution are handled in the `_handleTokenTransfer` function. It determines the net inbound or outbound transfer amount for each token, performs conversions between wrapped and native assets as needed, and executes the transfers. At the end of the function, a balance check is performed to convert any remaining native assets into wrapped assets. This is shown below:

```
function _handleTokenTransfer(...) internal {
    // ...
    // Depositing WETH
    if (address(this).balance > 0) {
        weth.deposit{value: address(this).balance}();
    }
}
```

There are multiple checks in the contract to ensure that native asset value cannot be transferred into the contract unless a wrapped native asset token is supported. These checks are present in all deposit, order, and fallback receive functions. However, an attacker can bypass these protections by using the `selfdestruct` opcode to forcefully transfer native assets to the contract. If this is done to a market contract that does not support native assets, the balance check will still execute. Assuming the attacker has forcefully deposited native assets using `selfdestruct`, the balance will read as non-zero, and the market will attempt to call `deposit` on the `weth` address variable. The `weth` variable is defined below:

```
IWETH weth = is_token_x_weth ? IWETH(address(token_x)) : IWETH(address(token_y));
```

A market that does not support native assets would have regular ERC20 tokens for `token_x` and `token_y`, so the `weth` variable would be an address that does not fully support the `IWETH` interface—particularly the `deposit` function. With native assets present on the contract, the check will still execute and the `deposit` call will fail, causing the transaction to revert.

The `_handleTokenTransfer` function is an internal function used by many other external functions. As a result, if native assets are forcefully sent to a market that does not support them, the following external functions will permanently revert: `placeOrder`, `placeMarketOrderWithValue`, `claimOrder`, and `batchClaim`. This means that users cannot make, claim, or cancel orders, effectively halting the market.

Recommendation(s): Consider adding an additional requirement to the balance check to ensure that `supports_native_eth` is true. This will ensure that the `weth.deposit` call only executes when the `weth` token is guaranteed to support the `deposit` function.

Status: Fixed

Update from the client: Fixed in commit `51067f20`.

6.3 [Info] ERC20 Permit support

File(s): `src/OnchainCLOB.sol`

Description: The `OnchainCLOB` contract supports tokens with the EIP-2612 standard, allowing users to provide signatures for allowances. There are two other notable permit implementations which are not supported by the `OnchainCLOB` contract: The Uniswap Permit2 and the DAI-style permit.

```
// EIP-2612
permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s)
// Permit2
permit(address owner, PermitSingle memory permitSingle, bytes calldata signature)
// DAI-style
permit(address holder, address spender, uint256 nonce, uint256 expiry, bool allowed, uint8 v, bytes32 r, bytes32 s)
```

Recommendation(s): No changes are required, this finding simply notes that there may be some permit based tokens which have a different standard to what is supported by the `OnchainCLOB` contract. When creating a new market for ERC20 tokens, consider their permit style.

Status: Acknowledged

Update from the client: No changes are required. We support EIP-2612 permits and will add other standards like Permit2 if needed.

6.4 [Info] Order claiming allows token transfers and is open by default

File(s): src/OnchainCLOB.sol

Description: There are two functions that can be used to claim an order: `claimOrder` and `batchClaim`. The `claimOrder` function is designed to be called by a user, using the `msg.sender` as the order owner. The `batchClaim` function allows an array of addresses to be passed, meaning that the caller does not have to be the order owner in order to claim an order on behalf of another user. `batchClaim` will call the internal `_calculateAndSendOrderPayout` function with the `transfer_tokens` argument hardcoded to `true`, as shown below:

```
function batchClaim(address[] memory addresses, ...) external ... {
    // ...
    for (uint i = 0; i < addresses.length; i++) {
        _calculateAndSendOrderPayout(addresses[i], order_ids[i], only_claim, true);
    }
}
```

The `OnchainCLOB` contract allows for two ways to manage assets when creating and claiming orders. Users can deposit and withdraw tokens on each order to maintain custody of their assets, or, for a more gas-efficient approach which could suit entities placing frequent orders, it is possible to leave tokens in the contract and use internal accounting. By using internal accounting, token transfers are not required, which can save gas costs for these high-frequency traders.

A user can only have their order claimed on their behalf by another address if their `claimable` status is set to `true`, which is the default state when their trader ID is created. This makes it possible to claim tokens on behalf of another user and forcefully move the tokens into the user's account. For traders frequently placing orders and making use of the internal accounting system to save gas, this opens the opportunity to "grief" these power users by forcing them to manually deposit the tokens again, requiring the user to submit token approvals and increasing their gas costs while interacting with the protocol.

Recommendation(s): Given that the default behavior of allowing orders to be claimable suits general users, consider allowing power users to have granular control over whether their tokens should be transferred or remain in the contract during a claim.

Status: Fixed

Update from the client: User can set desired behavior with function `setTraderConfig`. We removed function `batchClaim`, supposed using `batchExecute` with calls of `claimOrder` instead. In `claimOrder` if msg.sender != order_owner contract will ignore provided params.

6.5 [Info] Post-only orders can be griefed via front-running

File(s): src/OnchainCLOB.sol

Description: The `_placeOrder(...)` function allows users to submit `post-only` orders. These orders are designed to be passive, adding liquidity to the order book without executing against existing orders. The contract enforces this by checking if the order's price would cross the spread. If it does, the order is considered aggressive, and the function returns without placing it.

An attacker can exploit this mechanism by front-running a user's `post-only` order. By observing the transaction in the mempool, the attacker can submit their own small order that shifts the best price on the book just enough to make the victim's subsequent `post-only` order appear aggressive.

```
// ...
if (post_only) {
    bool aggressive_trade = false;
    if (isAsk) {
        // @audit An attacker can front-run to place a bid, making this condition true.
        //        Same applies for ask
        if (price_id <= bidTrie.best_offer()) {
            aggressive_trade = true;
        }
    }
    // @audit If front-run, aggressive_trade becomes true.
    if (aggressive_trade) {
        if (msg.value > 0) {
            _sendETH(msg.sender, msg.value);
        }
        // ...
        // @audit The function returns, causing the user's order to fail.
        return execution_result;
    }
}
// ...
```

When the victim's transaction is executed, the `aggressive_trade` check will pass, causing the function to return early. The user's order is never placed, their transaction effectively becomes a no-op, and they are forced to waste gas.

Recommendation(s): Consider documenting the behavior clearly for users intending to use `post_only` orders.

Status: Fixed

Update from the client: Fixed in commit 793e0b2d.

6.6 [Info] Proxy orders refund dust from `msg.value` to the order owner

File(s): src/OnchainCLOB.sol

Description: When a proxy places an order via `placeOrderByProxy(...)` and provides native ETH through `msg.value`, there might be a small amount of "dust" leftover. This occurs because the `msg.value` may not be a perfect multiple of the required token amount after scaling calculations. The protocol correctly calculates this dust but refunds it to the wrong party.

The `_handleTokenTransfer(...)` function handles the distribution of funds. It calculates the leftover ETH but sends it to the `client` address, which, in a proxy transaction, is the `order_owner`. The funds should be returned to `msg.sender`, the proxy who supplied the ETH in the first place.

```
function _handleTokenTransfer(
    address client, // @audit This is the `order_owner` in a proxy trade.
    // ...
    bool withdraw_as_native_eth
) internal {
    // ...
    if (msg.value > 0) {
        if (is_token_x_weth) {
            uint128 extra_x = (msg.value / scaling_factor_token_x).toUint128();
            uint256 actual_value = _convertToActualTokenXAmount(extra_x);
            // ...
            // @audit`rounded_value` is the dust from the proxy's msg.value.
            uint256 rounded_value = msg.value - actual_value;
            // @audit This dust amount is queued to be sent.
            actual_token_x_to_send = rounded_value;
        } else {
            // ... similar logic for token Y
        }
    }

    // ...

    // actual erc20 transactions
    if (actual_token_x_to_send > 0) {
        if (supports_native_eth && is_token_x_weth) {
            if (withdraw_as_native_eth) {
                // @audit Dust is sent to `client` (the order_owner), not msg.sender (the proxy).
                _sendETH(client, actual_token_x_to_send);
            } else {
                token_x.safeTransfer(client, actual_token_x_to_send);
            }
        }
        // ...
    }
    // ...
}
```

This leads to a situation where the proxy consistently loses a small amount of ETH on every native currency trade, while the `order_owner` unexpectedly receives these small refunds.

Recommendation(s): Consider updating the logic to correctly refund the dust amount to the supplier of the order.

Status: Acknowledged

Update from the client: The described behavior is the intended design. The logic is based on the protocol's expectation that all funds provided to the `placeOrderByProxy` function originate from the client (`order_owner`). Therefore, any residual dust from the transaction is correctly refunded to the source of the funds, the `order_owner`.

6.7 [Info] The function `getAccumulatedFees` returns a scaled value

File(s): `src/OnchainCLOB.sol`

Description: The function `getAccumulatedFees` is an external view function intended to provide information about the total fees collected by the protocol. However, the function returns the raw storage value of `accumulated_fees`, which is a scaled-up integer. It does not scale this value down by the `wad` factor (`1e18`) to represent the actual token amount.

```
function getAccumulatedFees() external view returns (uint256) {
    return (accumulated_fees - 1);
}
```

This is inconsistent with how fees are handled internally. For instance, the `_transferFees()` function correctly divides the `accumulated_fees` by `wad` to get the real amount of fees to be distributed.

```
function _transferFees() internal {
    // @audit The value is correctly scaled down here.
    uint256 total_fees = (accumulated_fees - 1) / wad;
    accumulated_fees -= total_fees * wad;
    // ...
}
```

As a result, anyone that relies on `getAccumulatedFees()` will receive a a misleading value, potentially causing incorrect calculations, confusion or misrepresentation of the protocol's performance.

Recommendation(s): Consider changing `getAccumulatedFees` to return the actual accumulated fee amount instead of the scaled amount.

Status: Fixed

Update from the client: Fixed in commit `759a74eb`. We have fixed the issue as recommended by returning the actual token amount after proper scaling. However, we also preserved the raw value for transparency and backward compatibility. Both values are now returned to ensure external integrations can properly work with the fee data.

6.8 [Best Practices] Function visibilities can be stricter

File(s): `src/*`

Description: Many functions in the `OnchainCLOB` and `Trie` contracts have a visibility of `public` even though they are not called internally by the contract. A visibility of `external` would be more appropriate as these functions will only be called by other contracts. The list of functions are presented below:

```
Trie.removeOrder
Trie.claimExecuted
Trie.getOrderInfo
Trie.executeRight
Trie.previewExecuteRight
OnchainCLOB.initialize
OnchainCLOB.placeOrder
OnchainCLOB.placeOrderByProxy
OnchainCLOB.placeMarketOrderWithTargetValue
OnchainCLOB.claimOrder
OnchainCLOB.depositTokens
```

Recommendation(s): The most accurate function visibility should be used where possible, consider adjusting the visibility of the functions listed above from `public` to `external`.

Status: Fixed

Update from the client: Fixed in the following commits: `a6c0645b`, `7b8ce220`, and `0fcf7b3c`.

6.9 [Best Practices] Unnecessary/unused imports

File(s): src/*

Description: There are some unnecessary and unused imports that can be removed from some contracts. Removing unnecessary code helps to improve readability and code clarity. The relevant imports are listed below:

```
WatchDogFactory.sol
    Ownable2Step    (Not used)
    Errors          (Not used)
```

Recommendation(s): Consider removing the unused imports.

Status: Fixed

Update from the client: Fixed in commit 7896084b.

6.10 [Best Practices] Unused errors

File(s): src/Errors.sol

Description: The file Errors.sol defines all errors used my multiple Hanji contracts. While many of these errors are used, there are some which are not used by any contracts. The following list presents the unused errors:

```
error ArrayLengthMismatch();
error CommissionParamTooHigh();
error Disabled();
error FractionalNumbersNotAllowed();
error NotImplementedYet();
error PriceExceedsMaximumAllowedValue();
error UnknownOrderId();
error UnknownTrader();
```

Recommendation(s): Consider removing the unused errors.

Status: Fixed

Update from the client: Fixed in commit 7896084b.

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 Hanji's documentation

The `Hanji` team provided an overview of the core system components during our meetings, presenting a clear explanation of the intended functionalities and addressing questions raised by the Nethermind Security team. These sessions offered valuable insights and ensured a comprehensive understanding of the project's technical foundations.

As supporting resources, the README outlined the structure of the on-chain Central Limit Order Book implementation, with the main logic contained in `OnchainLOB.sol` and order management built on the `TrieLib.sol` prefix tree. The documentation described the key order placement and management functions, token deposit and withdrawal operations, fee handling, and configuration options, along with the Watch Dog safety mechanism for monitoring chain stability. Build and testing instructions were also included to support development and verification of the contracts.

8 Test Suite Evaluation

8.1 Compilation Output

```
user@machine onchain-clob-contracts % forge build --via-ir --optimize
[.] Compiling...
[.] Compiling 115 files with Solc 0.8.30
[.] Solc 0.8.30 finished in 51.47s
```

8.2 Tests Output

```
Ran 3 tests for test/Allocator.t.sol:AllocatorTest
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 345.42µs (120.96µs CPU time)
Ran 5 tests for test/ZeroAddresses.t.sol:ZeroAddressesTest
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 1.20ms (211.38µs CPU time)
Ran 2 tests for test/WatchDog.t.sol:WatchDogTest
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 974.54µs (591.46µs CPU time)
Ran 3 tests for test/Governance.t.sol:GovernanceTest
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 1.16ms (139.67µs CPU time)
Ran 3 tests for test/SelfDestructAttack.t.sol:WETHDepositVulnerabilityPoCTest
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 1.17ms (263.29µs CPU time)
Ran 4 tests for test/MaxCommission.t.sol:MaxCommissionTest
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.17ms (282.88µs CPU time)
Ran 11 tests for test/Pausable.t.sol:PausableTest
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 1.97ms (830.08µs CPU time)
Ran 2 tests for test/DepositUseCases.t.sol:DepositUseCasesTest
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 1.60ms (734.54µs CPU time)
Ran 8 tests for test/NativeETHSupportForTokenX.t.sol:NativeETHSupportForTokenXTest
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 1.37ms (506.25µs CPU time)
Ran 8 tests for test/SafeTransferFrom.t.sol:SafeTransferFromTest
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 1.20ms (366.34µs CPU time)
Ran 2 tests for test/BatchClaim.t.sol:BatchClaimTest
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.72ms (1.38ms CPU time)
Ran 2 tests for test/FactoriesBasic.t.sol:FactoryTest
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 951.46µs (526.87µs CPU time)
Ran 8 tests for test/NativeETHSupportForTokenY.t.sol:NativeETHSupportForTokenYTest
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 1.34ms (495.75µs CPU time)
Ran 8 tests for test/Reentrancy.t.sol:ReentrancyTest
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 4.88ms (2.87ms CPU time)
Ran 8 tests for test/FeesRounding.t.sol:FeesRoundingTests
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 5.05ms (3.93ms CPU time)
Ran 17 tests for test/Erc20Permit.t.sol:Erc20PermitTest
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 7.51ms (7.20ms CPU time)
Ran 4 tests for test/MarketOrderWithTargetValue.t.sol:MarketOrderWithTargetValue
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.93ms (721.63µs CPU time)
Ran 10 tests for test/LobEvents.t.sol:LobTest
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 3.08ms (700.67µs CPU time)
Ran 21 tests for test/MarketMaker.t.sol:MarketMakerTest
Suite result: ok. 21 passed; 0 failed; 0 skipped; finished in 5.93ms (2.89ms CPU time)
Ran 7 tests for test/OnchainLOB.t.sol:LOBTest
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 30.51ms (2.79ms CPU time)
Ran 19 tests for test/Upgradeable.t.sol:UpgradeableTest
Suite result: ok. 19 passed; 0 failed; 0 skipped; finished in 31.56ms (2.62ms CPU time)
Ran 5 tests for test/LOBViews.t.sol:LOBViewsTest
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 79.13ms (79.15ms CPU time)
Ran 32 tests for test/Trie.t.sol:TrieUnitTest
Suite result: ok. 32 passed; 0 failed; 0 skipped; finished in 134.97ms (135.27ms CPU time)
Ran 2 tests for test/TradeWithLogComplexity.t.sol:TradeWithLogComplexityTest
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 1.62s (26.89ms CPU time)
Ran 2 tests for test/TradeWithLogComplexityWithFixedPrice.t.sol:TradeWithLogComplexityWithFixedPriceTest
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 1.83s (19.04ms CPU time)
Ran 49 tests for test/HanjiLOBUintTest.t.sol:OnchainCLOBUintTest
Suite result: ok. 49 passed; 0 failed; 0 skipped; finished in 2.09s (3.46ms CPU time)
Ran 4 tests for test/PriceOrderQueue.t.sol:PriceOrderQueueTest
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 2.09s (4.54s CPU time)
Ran 3 tests for test/AllValidPricesPackUnPack.t.sol:AllValidPricesTests
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 95.75s (117.90s CPU time)
Ran 12 tests for test/TrieLib.t.sol:TrieTestUnit
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 154.19s (154.24s CPU time)
```

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 conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to 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.

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

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.