Security Review Report
NM-0153 Carmine

NETHERMIND SECURITY

(Jan 8, 2024)

Contents

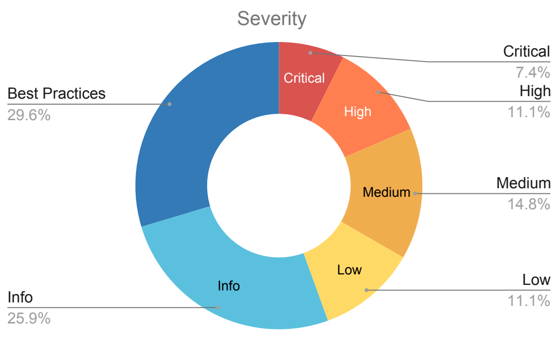
1 Executive Summary

This document outlines the security review conducted by Nethermind for the Carmine Options AMM. `Carmine` is a DeFi protocol revolutionizing options trading through its AMM. It offers a decentralized, automated platform for users by adopting a two-pool system for each asset pair, which manages call and put options distinctly. Liquidity providers are crucial, as they stake assets in these pools, taking the counter-position to user trades and maintaining a balance of both unlocked and locked capital. The protocol employs the Black-Scholes model for option pricing and features a volatility adjustment mechanism for post-trade changes. While LPs may enjoy high yields due to high demand, they also encounter short options and volatility risks. Therefore, careful risk management is required in response to fluctuating market conditions. Carmine's distinctive structure ensures liquidity and seamless operation, positioning it as a significant contributor to the evolving world of decentralized options trading.

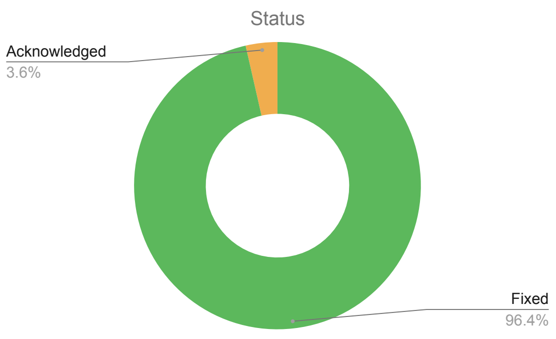
The audited code comprises 5878 lines of code in Cairo. The `Carmine` team has provided comprehensive documentation that explains the protocol architecture, their approach to liquidity pools, and the intricate mechanisms for option pricing and volatility updates.

The audit was performed using: (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contracts. **Along this document, we report** 28 points of attention, where two are classified as `Critical`, three are classified as `High`, four are classified as `Medium`, three are classified as `Low`, and sixteen are classified as `Informational` or `Best Practice`. 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 presents 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.



(a) distribution of issues according to the severity



(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (2), **High** (3), **Medium** (4), **Low** (3), **Undetermined** (0), **Informational** (8), **Best Practices** (8). **(b) Distribution of status: Fixed** (27), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Dec 21, 2023
Final Report	Jan 8, 2024
Methods	Manual Review, Automated Analysis
Repository	protocol-cairo1
Commit Hash	b4662ddd0a6f94c7cfeccb10a6b9720bec409163
Final Commit Hash	436e9d556426f6b55955ca3e2d6fb1fe7e70eef5
Documentation	Docs
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	erc20_interface.cairo	82	3	3.7%	5	90
2	lib.cairo	51	0	0.0%	1	52
3	amm_interface.cairo	219	1	0.5%	10	230
4	types/option_.cairo	262	19	7.3%	62	343
5	types/pool.cairo	66	5	7.6%	15	86
6	types/basic.cairo	14	0	0.0%	5	19
7	amm_core/trading.cairo	328	108	32.9%	63	499
8	amm_core/helpers.cairo	277	103	37.2%	94	474
9	amm_core/state.cairo	418	169	40.4%	118	705
10	amm_core/constants.cairo	36	12	33.3%	15	63
11	amm_core/amm.cairo	551	32	5.8%	90	673
12	amm_core/options.cairo	616	261	42.4%	140	1017
13	amm_core/liquidity_pool.cairo	336	119	35.4%	104	559
14	amm_core/oracles/agg.cairo	20	8	40.0%	4	32
15	amm_core/oracles/oracle_helpers.cairo	9	1	11.1%	6	16
16	amm_core/oracles/pragma.cairo	177	51	28.8%	34	262
17	amm_core/pricing/option_pricing.cairo	1417	72	5.1%	69	1558
18	amm_core/pricing/option_pricing_helpers.cairo	680	66	9.7%	69	815
19	amm_core/pricing/fees.cairo	26	10	38.5%	8	44
20	tokens/my_token.cairo	71	22	31.0%	21	114
21	tokens/option_token.cairo	134	35	26.1%	35	204
22	tokens/lptoken.cairo	88	30	34.1%	25	143
	Total	**5878**	**1127**	**19.2%**	**993**	**7998**

3 Summary of Issues

	Finding	Severity	Update
1	Incorrect calculation of locked capital because of using the wrong variable	Critical	Fixed
2	Incorrect calculation of locked capital when burning short options	Critical	Fixed
3	Calculation of pool expired position value skips option at index 0	High	Fixed
4	Liquidity pool value is incorrectly calculated due to the incorrect assumption that options are sorted by maturity	High	Fixed
5	Shadowing `usable_index` in the function `append_to_available_options(...)` permanently locks funds in old options	High	Fixed
6	Incorrect check for maximum liquidity pool balance	Medium	Fixed
7	Owner can overwrite existing option LP tokens	Medium	Fixed
8	Pool's position is incorrectly calculated	Medium	Fixed
9	Possible sandwich attack allows liquidity providers to earn fees without any risk	Medium	Fixed
10	Function `expire_option_token(...)` expires incorrect option side for the pool	Low	Fixed
11	Function `pow(...)` returns early for an edge case	Low	Fixed
12	Users cannot deposit or withdraw liquidity if an option is close to expiring	Low	Fixed
13	Incorrect assertion statement	Info	Fixed
14	Incorrect value of `capital_transfered` emitted in events	Info	Fixed
15	Loss of precision when converting from new types to `legacy` types could cause incorrect storage writes	Info	Fixed
16	Only governance can halt trading, defeating its purpose	Info	Fixed
17	Unnecessary logic in the function `expire_option_token_for_pool(...)`	Info	Fixed
18	Unused variables	Info	Fixed
19	View method can change the contract state	Info	Fixed
20	Wrong storage type conversion	Info	Fixed
21	Inconsistent check of return value for `transfer(...)` and `transferFrom(...)` functions	Best Practices	Fixed
22	Incorrect comments	Best Practices	Fixed
23	Merge two consecutive checks into one	Best Practices	Acknowledged
24	Unnecessary `Option` type return	Best Practices	Fixed
25	Unnecessary check of contract address and caller	Best Practices	Fixed
26	Unnecessary type casting	Best Practices	Fixed
27	Unnecessary write to storage during migration	Best Practices	Fixed
28	Use standard traits for type conversions	Best Practices	Fixed

4 System Overview

`Carmine` offers a decentralized, automated platform for users by adopting a two-pool system for each asset pair, which distinctly manages call and put options. Liquidity providers take the counter-position to user trades and maintain a balance of both unlocked and locked capital. The protocol employs the Black-Scholes model for option pricing and features a volatility adjustment mechanism for post-trade changes. The protocol is conformed by a monolithic contract as its core, managing all the relevant operations like opening a trade, closing a trade, settling a trade, and depositing and withdrawing liquidity. This core contract interacts with other token contracts for minting and burning liquidity and option tokens; however, the state of each pool is kept in the core contract.

Even though all the mentioned operations are handled by only one contract, the code is organized in what we can call modules or libraries. The main modules of the system include:

a) **Liquidity Pool**
b) **Options**
c) **State**
d) **Trading**
e) **Pricing**
f) **Oracle**

Fig. 2 presents the interaction diagram of the contracts.

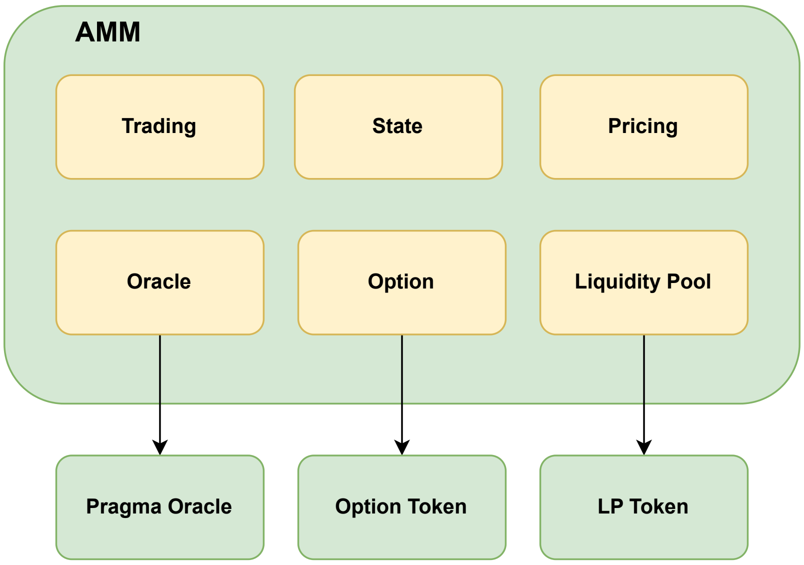


Fig. 2: Interaction Diagram of Contracts

The **Liquidity Pool** module contains the logic related to the operations modifying the liquidity pools state. This module includes functions for creating a liquidity pool for a new pair of assets and type of option, depositing or withdrawing liquidity to or from a specific liquidity pool, and settling the pool's positions in a particular trade.

The **Options** module will handle all the operations regarding Options. Options are defined by a pair of assets, the type of the option, the maturity time, and the strike price. The options will be created for a specific liquidity pool, defining with this the pair of assets and the option type. Furthermore, when adding a new option token, this will include information about the side of the trade; a token representing the **long** side will allow users to buy this kind of option, while a token representing the **short** side will allow the users to sell it. In this library, we can find functions for adding these new option tokens to the liquidity pools and minting and burning option tokens for or from the users.

The **State** module contains functions for reading and updating all the storage variables from the contract. Every time a method from a module needs to operate with the state of the contract, either for reading or writing operations, it will use the functions defined in the **State** library.

The **Trading** module aggregates the methods related to opening, closing, and settling trades. Opening a trade means buying or selling options from or to the AMM. Closing a trade will sell or buy back the options from the trade, but their price will be calculated again. Finally, settling a trade is called to realize any profit or loss from the opened trade; it can only be executed after the options have reached their maturity time.

In the **Pricing** module, we can find the implementation of the Black-Scholes model used for pricing the options.

The **Oracle** module contains the integration with different oracles. Oracles are needed to check the price of assets when computing the price of an option and when settling a trade to calculate the profit and loss for the traders. At the time of this review, the Carmine protocol relies solely on Pragma as an oracle.

The modules mentioned above are aggregated in an **AMM** contract. This contract contains the definition for the storage of the **AMM** and implements all the functions from its interface. These implementations are mainly wrappers to the functions from the described modules.

We can split the accounts that will interact with Carmine protocol into three different types of actors:

a) **Users:** This set will include regular users from the protocol. It mainly includes traders and liquidity providers. Bots or similar used to close positions for the liquidity pools are also accounted into this group. Accounts in this group are expected to execute operations like depositing and withdrawing liquidity, opening, closing, settling trades, and closing positions from the pool in order to unlock liquidity.

b) **Governance:** This set contains accounts with higher privileges in the protocol. These accounts are allowed to execute critical operations like creating liquidity pools, adding options to them, and halting trading operations. Incorrect use of the privileges from this actor could cause irreversible damage to the protocol. It is highly recommended to only provide these privileges to a reduced number of accounts that are secured with the safest measures.

c) **Admin:** The members of this group are allowed to halt the trading operations from the protocol. They are expected to act as a contingency response if something unexpected occurs. Even though this set of users does not have the same amount of privileges as governance, it is highly recommended to only provide these privileges to a reduced number of accounts that are secured with the safest measures.

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 an attacker will uncover and exploit the finding. 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 formally pass to the client;

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 [Critical] Incorrect calculation of locked capital because of using the wrong variable

File(s): `amm_core/options.cairo`

Description: In the else-branch of the function `_burn_option_token_short(...)`, the user burns a short option, thereby increasing the pool's short. Consequently, the pool needs to lock more capital by adding `option_size_in_pool_currency` to the currently locked capital. However, when calculating `new_locked_capital`, the function adds to `current_unlocked_capital_u256` instead of `current_locked_capital_u256`. This is incorrect and will also cause subsequent updates of locked capital to be erroneous. The result of this could be that options can't be settled due to insufficient locked capital, or liquidity providers might withdraw more than the current free capital.

```
let current_unlocked_capital_u256 = get_unlocked_capital(lptoken_address);

// Update locked capital
// @audit Should be `current_locked_capital_u256` instead of `current_unlocked_capital_u256`
let new_locked_capital = current_unlocked_capital_u256 + option_size_in_pool_currency;

assert(
    option_size_in_pool_currency <= current_unlocked_capital_u256,
    'BOTS - not enough capital'
);
```

Recommendation(s): Correct the code by changing `current_unlocked_capital_u256` to `current_locked_capital_u256`.

Status: Fixed.

Update from the client: Fixed in audit_fixed branch: 9ebe00f062dda8c19c12c669f959cee638f30a1d

6.2 [Critical] Incorrect calculation of locked capital when burning short options

File(s): `amm_core/options.cairo`

Description: In the function `_burn_option_token_short(...)`, traders burn their short options, which increases the pool's short position. When this happens, more capital from the liquidity pool should be locked. However, the current implementation decreases the liquidity pool's locked capital.

As a result, the locked capital calculation is incorrect, locking less capital than required. This miscalculation allows liquidity providers to withdraw funds when they shouldn't, leading to situations where options cannot be settled due to insufficient locked capital. The code snippet below highlights the issue:

```
let strike_price_u256: u256 = toU256_balance(strike_price, quote_address);
let capital_to_be_locked = convert_amount_to_option_currency_from_base_uint256(
    increase_short_position_by, option_type, strike_price_u256, base_address
);

// @audit Should add more locked capital instead of reducing
let new_locked_capital = current_locked_capital_u256 - capital_to_be_locked;
```

Recommendation(s): Correct the calculation of new locked capital in the function `_burn_option_token_short(...)`.

Status: Fixed.

Update from the client: Fix in audit_fixes branch: 79446d69f752d928dc6dcd7abcc0fda553292c57. This bug was a typo that was not caught during rewriting the Cairo 0 to this version. The Cairo 0 version of the smart contracts do not include this bug. I also added a test for this particular branch of the function: 44237750b15f5349eea42247309a91082c56c079

6.3 [High] Calculation of pool expired position value skips option at index 0

File(s): `amm_core/liquidity_pool.cairo`

Description: The function `get_value_of_pool_expired_position(...)` calculates the total value of pool expired positions using a backward loop from the last index. However, it fails to consider the value of the option at index 0. As indicated in the code snippet, if `ix == 0`, the loop breaks prematurely, excluding the option's value from the total value.

This oversight could result in incorrect calculations of LP token value during minting or burning. This could potentially lead to liquidity providers receiving more LP tokens when minting and fewer underlying tokens when burning.

```
loop {
    let option = get_available_options(lptoken_address, ix);
    assert(option.sum() != 0, 'GVoEO - opt sum zero');
    ...

    // @audit Value of option at index 0 is skipped
    if ix == 0 {
        break;
    }
    ix -= 1;
    ...

    pool_pos += option.value_of_position(option_position);
};

```

Recommendation(s): Correct the logic to break the loop after adding the value of the position when `ix == 0`.

Status: Fixed.

Update from the client: Fix is part if this commit: 5889f8b10dcdd5c6bed87d24753582e69a5fcb5a. Lines 155-164.

Update from Nethermind: The fix is good, but we have a suggestion. Since the function no longer breaks early and no longer has `LOOKBACK`, consider using a forward loop instead of a backward loop to simplify the code.

Update from the client: Forward loop implemented: bbf891a5cad43aeaba25f7374898c93a87129584

6.4 [High] Liquidity pool value is incorrectly calculated due to the incorrect assumption that options are sorted by maturity

File(s): `amm_core/liquidity_pool.cairo`

Description: The function `get_value_of_pool_expired_position(...)` incorrectly calculates the total value of expired pool positions. It iterates backward, wrongly assuming options are sorted by maturity. The function that adds options does not verify this. This issue is evident in the code snippet below.

Consequently, the total pool value calculation might exclude the values of some expired pools because the loop breaks earlier than it should. This discrepancy could lead to incorrect LP token calculations, resulting in liquidity providers receiving more or fewer LP tokens than anticipated.

```
fn get_value_of_pool_expired_position(lptoken_address: LPTAddress) -> Fixed {
    let LOOKBACK = 24 * 3600 * 7 * 8;
    // ^ Only look back 8 weeks, all options should be long expired by then
    let now = get_block_timestamp();
    let last_ix = get_available_options_usable_index(lptoken_address);
    ...

    // @audit Loop backward
    let mut ix = last_ix - 1;
    let mut pool_pos: Fixed = FixedTrait::from_felt(0);

    loop {
        let option = get_available_options(lptoken_address, ix);
        ...
        // @audit Breaking the loop as soon as the current option has not yet expired
        if (option.maturity >= now) {
            break; // Option is not yet expired
        };
        ...
        pool_pos += option.value_of_position(option_position);
    };

    pool_pos
}
```

Recommendation(s): Consider adding validation to ensure options are sorted by maturity in function `add_option(...)`.

Status: Fixed.

Update from the client: Firstly fixed here: 5889f8b10dcdd5c6bed87d24753582e69a5fcb5a (fixed the check for `maturity >= now`) and later removed `LOOKBACK` too: 74821eff63c8ee3f2b241d0dcb3e0cc695a63bdc.

Update from Nethermind: The fix is good, but we have a suggestion. Since the function no longer breaks early and no longer has `LOOKBACK`, consider using a forward loop instead of a backward loop to simplify the code.

Update from the client: Forward loop implemented: bbf891a5cad43aeaba25f7374898c93a87129584

6.5 [High] Shadowing `usable_index` in the function `append_to_available_options(...)` permanently locks funds in old options

File(s): `amm_core/state.cairo`

Description: All options in the protocol are stored in a list. In this list, `usable_index` indicates the next empty position. When migrating to Cairo 1, it transfers all the old options to a new list, records the last `usable_index`, and then proceeds.

However, `usable_index` is shadowed in the inner scope (within the if block). Consequently, when it exits the block, the value of `usable_index` remains `0`. The contract then considers the migration step complete and appends new options to the list. However, because `usable_index` begins from 0, it overwrites all the old options, making them inaccessible to users.

```
fn append_to_available_options(option: Option_, lptoken_address: LPTAddress) {
    let mut state = AMM::unsafe_new_contract_state();

    // Read storage var containg the usable index
    let usable_index = get_available_options_usable_index(lptoken_address);

    // In this case we need to migrate the old options to the new storage var
    // since this storage var was introduced and is used only in c1 version

    // @audit `usable_index` stays 0 after exiting the if block (variable shadowing)
    if usable_index == 0 {
        let usable_index = migrate_old_options(lptoken_address, 0);
    }

    state.new_available_options.write((lptoken_address, usable_index), option);

    // Increase the usable index in available options
    state.new_available_options_usable_index.write(lptoken_address, usable_index + 1);
}
```

Recommendation(s): Do not shadow `usable_index` inside the if block.

Status: Fixed.

Update from the client: Decided to remove the migrating functionality altogether: 636af75a91250f542582d413fd7cebf0b82381da

6.6 [Medium] Incorrect check for maximum liquidity pool balance

File(s): `amm_core/liquidity_pool.cairo`

Description: Each liquidity pool has a maximum total balance that limits the amount all liquidity providers can deposit into a given pool. This limit is examined in the function `deposit_liquidity(...)` after calculating the pool's new balance post-deposit. However, the current check utilizes the `current_balance` instead of the newly calculated `new_balance`. This is incorrect and could allow the pool to exceed its defined maximum.

```
// Update the lpool_balance after the mint_amount has been computed
// (get_lptokens_for_underlying uses lpool_balance)
let current_balance = get_lpool_balance(lptoken_address);
let new_balance = current_balance + amount;
set_lpool_balance(lptoken_address, new_balance);

let max_balance = get_max_lpool_balance(lptoken_address);

// @audit Should check `new_balance` instead of `current_balance`
assert(current_balance <= max_balance, 'Lpool bal exceeds maximum');
```

Recommendation(s): Use `new_balance` for the check instead of `current_balance`.

Status: Fixed.

Update from the client: Fix in audit_fixes branch: fc251bb08e7fc15b566e85eb5320a62dde27ff18

6.7 [Medium] Owner can overwrite existing option LP tokens

File(s): `amm_core/liquidity_pool.cairo`

Description: Function `add_lptoken(...)` adds new LP tokens according to the specified option. However, the owner can add a new LP token to overwrite the existing LP token, which can cause accounting problems and centralizations.

```
fn add_lptoken(
    quote_token_address: ContractAddress,
    base_token_address: ContractAddress,
    option_type: OptionType,
    lptoken_address: LPTAddress,
    pooled_token_addr: LPTAddress,
    volatility_adjustment_speed: Fixed,
    max_lpool_bal: u256
) {
    // ...

    set_lptoken_address_for_given_option( // @audit-issue: Can overwrite existing option lp token
        quote_token_address, base_token_address, option_type, lptoken_address
    );
    // ...
}
```

In case of mistakenly overwriting existing LP tokens, the owner won't be able to undo their changes because the following supply check will always revert.

```
let supply_lpt = IERC20Dispatcher { contract_address: lptoken_address }.totalSupply();
assert(supply_lpt == 0, 'LPT minted != 0');
```

Recommendation(s): Check whether an LP token was already added for this option configuration.

Status: Fixed.

Update from the client: Fix in audit_fixes branch: 60b4e11b13bb092918724fd95d844dad30274c6f

6.8 [Medium] Pool's position is incorrectly calculated

File(s): `amm_core/liquidity_pool.cairo`

Description: Calculating the pool's position is integral to operations like depositing or withdrawing liquidity. To compute this value, the valuation method involves iterating over all unsettled options to aggregate the pool's position. However, an issue arises when an option for a specific trade side (e.g., `SHORT`) is not registered, even though the pool holds a position on that side of the trade. This omission leads to an inaccurate calculation of the pool's total value, as these positions are ignored in the valuation process.

Consider a scenario with an `ETH/USDC Call` pool where users can only open `LONG` positions. The pool will take the `SHORT` position. If no option is registered for the `SHORT` side, the pool's valuation during the position calculation does not account for this position, leading to an undervalued pool.

Recommendation(s): All the options are registered by governance; ensure that both sides of the trade are always registered for every option definition. Consider adding this requirement to the smart contract code.

Status: Fixed.

Update from the client: Fix in audit_fixed branch: 4dc6010b31e76c67ccdbf57d8ab6add1ef40390e

6.9 [Medium] Possible sandwich attack allows liquidity providers to earn fees without any risk

File(s): `amm_core/liquidity_pool.cairo`

Description: The Carmine AMM has two main actors: the trader and the liquidity provider. Liquidity providers deposit their funds into the liquidity pool and take the opposing side of traders' trade. To compensate, traders pay a fee to the liquidity providers on all trades. This incentivizes liquidity providers to deposit their funds and take the risk. However, the Carmine AMM allows liquidity providers to deposit and withdraw instantly, which opens the possibility for a sandwich attack. This attack involves depositing liquidity and then withdrawing it immediately before and after a trader makes a trade.

Consider this scenario:

1. The pool currently has `100 ETH` of unlocked capital. Let's say Bob is a trader who will open a position requiring the pool to lock `90 ETH`, which means he will need to pay a `0.1 ETH` fee;
2. Alice, as an MEV bot, sees this opportunity. She executes a sandwich attack by depositing `1000 ETH`, becoming the largest depositor and collecting most of the `0.1 ETH` fee that Bob pays;
3. After doing this, Alice immediately withdraws all her funds. Since Bob's position only locked `90 ETH`, Alice can take everything back;

As you can see, all three transactions can occur within a single block. Afterward, Alice receives almost all of the liquidity pool's fees without taking any risk because she has already withdrawn all her funds.

Recommendation(s): Consider not allowing withdrawal of the liquidity instantly after depositing.

Status: Fixed.

Update from the client: Fix in audit_fixed branch: 5afb766e26fbe4ddebb044c7900bdc27aa0c14ca. Later down the line we are likely to use a third-party service that will work as a middleware between the user and Carmine AMM and that will more generically identify malicious transactions.

Update from Nethermind: Liquidity providers can transfer the LP token to another address after depositing liquidity, then withdraw from that address to bypass the `SandwichGuard`.

Update from the client: `SandwichGuard` moved to LP token: c3147e9e94d607f891c2862e0915d0ac6c5f5f29. It is now impossible to transfer/mint/burn token if any mint/burn occurred within the same block.

6.10 [Low] Function `expire_option_token(...)` expires incorrect option side for the pool

File(s): `amm_core/options.cairo`

Description: The function `expire_option_token(...)` expires a user's option tokens. It should first expire the option for the pool, which means settling any capital locked on the pool's side. However, it initially checks if the current pool position of `TRADE_SIDE_SHORT` is non-zero, but then settles the option token for `option_side` instead of `TRADE_SIDE_SHORT`. Although Carmine has a keeper bot that expires all options upon maturity, handling this at the smart contract level would be better. The code snippet below demonstrates the issue.

```
let current_pool_position = get_option_position(
    lptoken_address, TRADE_SIDE_SHORT, maturity, strike_price
);

// @audit `TRADE_SIDE_SHORT` is checked but `option_side` is settled
if (current_pool_position != 0) {
    expire_option_token_for_pool(lptoken_address, option_side, strike_price, maturity,);
}
// Check that the pool's position was expired correctly
let current_pool_position_2 =
    get_option_position( // FIXME this is called twice in the happy case
    lptoken_address, option_side, maturity, strike_price
);
assert(current_pool_position_2 == 0, 'EOT - pool pos not zero');
```

Recommendation(s): Consider expiring both option sides (long and short) of the pool before expiring users' options.

Status: Fixed.

Update from the client: We have changed the structure a little bit to avoid any misunderstandings in the future.
Here ff2cc826801f3e6cbd47666708b031b7dbc6bdf2 in the audit_fixes branch. Now, in case there is a user settling options before the pool does, the pool's position in a given option is settled first.

6.11 [Low] Function `pow(...)` returns early for an edge case

File(s): `amm_core/helpers.cairo`

Description: Method `pow(a: u128, b: u128)` calculates b power of a. However, there is no check for whether the base value is zero. 0^0 results are undefined in math, but in this case, it returns 1.

```
fn pow(a: u128, b: u128) -> u128 {
    let mut x: u128 = a;
    let mut n = b;

    if n == 0 { // @audit-issue: Zero power zero is undefined but, in this case, returns 1
        return 1;
    }
    // ...
}
```

Recommendation(s): Consider adding a check for parameter `a` is higher than zero.

Status: Fixed.

Update from the client: Fix in audit_fixed branch: a07065df27e8efec5039764d13c39676f4915465

6.12 [Low] Users cannot deposit or withdraw liquidity if an option is close to expiring

File(s): `types/option_.cairo`

Description: The smart contract's logic for computing the value of a pool's position during liquidity transactions (deposits or withdrawals) involves aggregating the value of unsettled options. This computation varies based on the option's maturity time. An issue arises for options nearing maturity, specifically those within a defined time frame before the maturity period ends. In these cases, the option's value cannot be determined, and the transaction will revert.

```
fn value_of_position(self: Option_, position_size: Int) -> Fixed {
    let current_block_time = get_block_timestamp();
    let is_ripe = self.maturity <= current_block_time;

    let position_size_cubit = fromU256_balance(position_size.into(), self.base_token_address);
    if is_ripe {
      ...
      // @audit - Compute value if the option has already reached maturity
    }
    // @audit - Compute value if the option has not reached maturity
    // Fail if the value of option that matures in 2 hours or less (can't price the option)
    let stop_trading_by = self.maturity - STOP_TRADING_BEFORE_MATURITY_SECONDS;
    // @audit - When this condition is false, the transaction reverts, making it impossible to deposit or withdraw
     ↪  liquidity
    assert(current_block_time <= stop_trading_by, 'GVoP - Wait till maturity');
    ...
    }
```

As can be seen in the previous code snippet, when the pool contains one or more options nearing their maturity (defined as within the `STOP_TRADING_BEFORE_MATURITY_SECONDS` period), the assertion checks `assert(current_block_time <= stop_trading_by)` makes the transaction revert. This behavior prevents transactions related to depositing or withdrawing liquidity, effectively freezing pool operations.

Recommendation(s): Consider either removing the check or correcting the documentation to prevent any misunderstanding.

Status: Fixed.

Update from the client: Fix in audit_fixed branch: 190cae3b205f1b45d4bcd747ccc3b5fa571792fb

6.13 [Info] Incorrect assertion statement

File(s): `amm_core/state.cairo`

Description: The following assertion error statement is incorrect. It should raise a statement including the strike price.

```
fn set_option_token_address(
    // ...
) {
    // ...
    strike_price.assert_nn_not_zero('sota - maturity <= 0');
    // ...
}
```

Recommendation(s): Consider changing the error statement.

Status: Fixed.

Update from the client: Fixed in audit_fixed branch: bee84f819b9f839005b180d403e9f757d4a3e06f

6.14 [Info] Incorrect value of `capital_transfered` emitted in events

File(s): `amm_core/options.cairo`

Description: In the `_mint_option_token_short(...)` function, the `TradeOpen` event is emitted with `capital_transferred` set to `premia_-including_fees_u256`. However, the actual amount of funds transferred in this function is `to_be_paid_by_user`. The code snippet below illustrates the issue. A similar issue also exists in the `_burn_option_token_short(...)` function.

```
let to_be_paid_by_user = option_size_in_pool_currency - premia_including_fees_u256;

emit_event(
    TradeOpen {
        caller: user_address,
        option_token: option_token_address,
        // @audit Incorrect value of `capital_transfered`
        capital_transfered: premia_including_fees_u256,
        option_tokens_minted: option_size.into()
    }
);
...

let transfer_res = IERC20Dispatcher { contract_address: currency_address }
    .transferFrom(user_address, curr_contract_address, to_be_paid_by_user);

assert(transfer_res, 'MOTS: unable to transfer premia');
```

Recommendation(s): Correct the value of `capital_transferred` in the event emissions.

Status: Fixed.

Update from the client: Fix in audit_fixed branch: 3f6e7ff18ba9715e83522f7840cff53cf44ce0fc

6.15 [Info] Loss of precision when converting from new types to `legacy` types could cause incorrect storage writes

File(s): `amm_core/state.cairo`

Description: The Carmine protocol integrates a `migration` mechanism to the protocol. The mechanism is designed to transition values to new storage positions while converting them to new types. A possible issue arises in the conversion of `strike_price` values from a higher-precision format (Q64.64) to a lower-precision format (Q64.61), leading to data truncation.

An example of this process is shown in the following code snippet.

```
   fn get_option_token_address(...) -> ContractAddress {
    ...
    // @audit - Conversion of `strike_price` may cause truncation
    let option_token_addr = state.option_token_address.read((..., strike_price.to_legacyMath()));
    ...
    // @audit - Potential incorrect update to new storage variable
    if !option_token_addr.is_zero() {
        ...
        set_option_token_address(..., option_token_addr);
        ...
    }
    ...
    // @audit - Reading from new storage variable
    let res = state.new_option_token_address.read((..., strike_price.mag));
    return res;
}
```

Truncation may occur when converting `strike_price` for fetching values from the old storage, potentially referencing a different option. This issue can lead to incorrect readings and unintended updates in the new storage variable.

Let's check an example; for simplification, let's use the types Q1.4 and Q1.1. If we try to fetch the option token address for an option with `strike_price = 1.337`, when this value is converted to Q1.1, the result will be `1.3`. If an option exists with the same parameters as the original and a `strike_price = 1.3`, this option will be incorrectly read, and the new storage variable will be updated.

Recommendation(s): Only governance can add new options to the liquidity pools. Thoroughly review the parameters of every option before it is deployed and ensure that parameters are compatible with legacy types.

Status: Fixed.

Update from the client: All the migrating functionalities were removed: 636af75a91250f542582d413fd7cebf0b82381da with legacy types removed here: 5ef83c22cadac0f5fd5a11422009be51cee83ab9

6.16 [Info] Only governance can halt trading, defeating its purpose

File(s): `amm_core/amm.cairo`

Description: A trading halt serves as a swift protective measure if abnormal behavior occurs within the protocol. For instance, if the token balance of the pool decreases rapidly. Temporarily halting trading allows the team to have time to investigate and rectify the issue. However, only the owner, set to the governance address, can enable a trading halt. This process requires proposing a transaction, waiting for a delay, and executing the transaction from the governance side. This delay defeats the purpose of the trading halt.

Recommendation(s): Consider allowing other addresses to halt trading, not just governance.

Status: Fixed.

Update from the client: Fix in audit_fixed branch: fb6a886a9f283a4495f676649824280d619292e5. Subsequent fix that a) made it impossible for allowed addresses to turn trading back again and b) makes it so that governance sets the permission for address to halt trading: 9972e000319883dc1f20fb3e52e836d51adfaed0 Also note that in one previous commit we fixed the discrepancy between annotations and code - if trading_halt is `true` then trading is halted (as opposed to before, when `true` meant that trading not halted).

6.17 [Info] Unnecessary logic in the function `expire_option_token_for_pool(...)`

File(s): `amm_core/liquidity_pool.cairo`

Description: The function `expire_option_token_for_pool(...)` contains an unnecessary check before setting the value of the option position. This check validates `opt_size_u256 <= current_pool_position`. However, these two values are return values of the function `get_option_position(...)` with the same input parameters, which means they will always be equal. Consequently, the check is unnecessary.

```
let current_pool_position = get_option_position(
    lptoken_address, option_side, maturity, strike_price
);

let new_pool_position = current_pool_position - option_size;

let opt_size_u256: u256 = option_size.into();

// @audit Unnecessary logic since opt_size_u256 and current_pool_position have the same value
assert(opt_size_u256 <= current_pool_position.into(), 'Opt size > curr pool pos');
```

Recommendation(s): Consider reviewing the intended behavior of the check.

Status: Fixed.

Update from the client: Fix in audit_fixes branch: 3715599e93684db0db5b450d8e3ab64515a8702e

6.18 [Info] Unused variables

File(s): `amm_core/liquidity_pool.cairo`, `amm_core/state.cairo`

Description: The codebase contains several declared but never used variables.

– The function `withdraw_liquidity(...)` defines a `unlocked_capital` variable that is not used;
– The function `expire_option_token_for_pool(...)` calculates `new_pool_position` but does not use it;
– The function `get_unlocked_capital(...)` fetches `state` but does not use it;
– The function `add_lptoken(...)` contains an unused input parameter `pooled_token_addr`;

Recommendation(s): Consider deleting unused variables.

Status: Fixed.

Update from the client: Fix in audit_fixed branch: cee3d21f819d436d56838f07fd563fcdde5a90cd variable in expire_option_token_for_pool removed here: 3715599e93684db0db5b450d8e3ab64515a8702e.
Unused argument in add_lptoken removed here: 7027053cfa6d7d86b8eb152c0bbc286fb6ef9ff3.

6.19 [Info] View method can change the contract state

File(s): amm_core/amm.cairo, amm_core/state.cairo

Description: In Starknet, there is no staticcall. So, every method can actually change the state of the contract. However, view method parameters start with the parameter self: @ContractState, which is a snap of the contract state, and write method parameters start with reference to the contract state. But this will not block the view method to change the contract state. Direct access to storage write methods and using unsafe_new_contract_state() will allow you to change the contract state in all kinds of methods.

```
fn get_option_token_address(
    self: @ContractState,
    lptoken_address: ContractAddress,
    option_side: OptionSide,
    maturity: u64,
    strike_price: Fixed,
) -> ContractAddress {
    // @audit-issue: The following function will change the contract state with `unsafe_new_contract_state`
    State::get_option_token_address(lptoken_address, option_side, maturity, strike_price,)
}
```

```
fn get_option_token_address(
    lptoken_address: LPTAddress,
    option_side: OptionSide,
    maturity: Timestamp,
    strike_price: Strike
) -> ContractAddress {
    let mut state = AMM::unsafe_new_contract_state(); // @audit-issue: This can be called in view calls too.
    // ...
    if !option_token_addr.is_zero() {
        // Write value to new storage var
        set_option_token_address(
            lptoken_address, option_side, maturity, strike_price, option_token_addr
        );
        // Set old storage var to zero
        state
            .option_token_address
            .write(
                (lptoken_address, option_side, maturity.into(), strike_price.to_legacyMath()),
                contract_address_try_from_felt252(0).expect('Cannot create addr from 0')
            ); // @audit-issue: State is changed.
        return option_token_addr;
    }
// ...
}
```

Similar issues exist in following functions: get_option_volatility(...), get_pool_volatility_adjustment_speed(...) and get_option_-position(...)

Recommendation(s): Consider developing a new method to read option token addresses for view calls.

Status: Fixed.

Update from the client: This was caused by migrating functionalities which were removed: 636af75a91250f542582d413fd7cebf0b82381da

6.20 [Info] Wrong storage type conversion

File(s): `amm_core/amm.cairo`

Description: Amm contracts cairo 0 implementations, storage is like the following

```
@storage_var
func option_position_(
    lptoken_address: Address, option_side: OptionSide, maturity: Int, strike_price: Int
) -> (res: Int) {
}
```

However, in new implementation `option_position_` is stored like following

```
// ...
option_position_: LegacyMap<(LPTAddress, OptionSide, Maturity, LegacyStrike), felt252>,
// @audit-issue: The old mapping value type is Int, not felt252
// ...
```

Since the `Int` type in Cairo 0 represents felts, it doesn't generate any risk, but using the same type is more accurate.

Recommendation(s): Consider storing as `Int` instead of `felt252`

Status: Fixed.

Update from the client: Decided to remove the migrating functionality altogether: 636af75a91250f542582d413fd7cebf0b82381da

6.21 [Best Practice] Inconsistent check of return value for `transfer(...)` and `transferFrom(...)` functions

File(s): `amm_core/liquidity_pool.cairo`

Description: It is a known best practice to check the result value from the `transfer(...)` and `transferFrom(...)` functions. Throughout the code, we could find multiple calls to the mentioned functions; however, their return values were only checked in some of those occurrences.

Recommendation(s): Consistently check the return value of `transfer(...)` and `transferFrom(...)` functions

Status: Fixed.

Update from the client: Fix in audit_fixed branch: 668158e0417bcd36ba305ba3bfe3b93fb0455c6e

6.22 [Best Practices] Incorrect comments

File(s): `amm_core/helpers.cairo`

Description: There's an error in the comment within the `fromU256_balance(...)` function. The value should be `2 64, not 261`. The code snippet below illustrates the issue.

```
fn fromU256_balance(x: u256, currency_address: ContractAddress) -> Fixed {
    // We will guide you through with an example
    // x = 1.2*10**18 (example input... 10**18 since it is ETH)
    // We want to divide the number by 10**18 and multiply by 2**64 to get Math64x61 number
    // But the order is important, first multiply and then divide, otherwise the .2 would be lost.

    // @audit Comment below should be 2**64 instead of 2**61
    // (1.2 * 10**18) * 2**61 / 10**18
    // We can split the 10*18 to (2**18 * 5**18)
    // (1.2 * 10**18) * 2**64 / (5**18 * 2**18)

    let decimals: u128 = get_decimal(currency_address).expect('fromU256 - decimals zero').into();

    _fromU256_balance(x, decimals)
}
```

Recommendation(s): Adjust the comment to prevent confusion when reading the code.

Status: Fixed.

Update from the client: Fixed in audit_fixed branch: e671a8d2dd5252d91965fb754fb197defaf06f16

6.23 [Best Practices] Merge two consecutive checks into one

File(s): amm_core/trading.cairo

Description: In the function `validate_trade_input(...)`, when a user wants to open a position, it's necessary to verify that the option hasn't expired and that the current block timestamp is before the option's stop trading timestamp. However, as shown in the code snippet below, since `STOP_TRADING_BEFORE_MATURITY_SECONDS > 0`, the first check will always fail if the second check fails. This indicates that only the second condition needs to be checked.

```
if open_position {
    // @audit These two checks below both check `current_block_time` to be less than something so they could be merged
    assert(current_block_time < maturity, 'VTI - opt already expired');
    assert(
        current_block_time < (maturity - STOP_TRADING_BEFORE_MATURITY_SECONDS),
        'VTI - Trading is no mo'
    );
}
```

Recommendation(s): Consider only maintaining the second assertion.

Status: Acknowledged

Update from the client: Decided to keep them both for better error reporting.

6.24 [Best Practices] Unnecessary `Option` type return

File(s): amm_core/helpers.cairo

Description: Function `get_decimals(...)` is supposed to return the decimal number of ERC20 tokens. However, this function returns `Option`, which is not the case.

```
fn get_decimal(token_address: ContractAddress) -> Option<u8> {
    if token_address == TOKEN_ETH_ADDRESS.try_into()? {
        return Option::Some(18);
    }

    if token_address == TOKEN_USDC_ADDRESS.try_into()? {
        return Option::Some(6);
    }

    assert(!token_address.is_zero(), 'Token address is zero');

    let decimals = IERC20Dispatcher { contract_address: token_address }.decimals();
    assert(decimals != 0, 'Token has decimals = 0');

    if decimals == 0 {
        ////////////////////////////////////////////////////////////////////////////////////////
        // @audit-issue: Unnecessary case, already reverted before.
        // There is no case to return Option::None. Use direct u8.
        ////////////////////////////////////////////////////////////////////////////////////////
        return Option::None(());
    } else {
        let decimals_felt: felt252 = decimals.into();
        return Option::Some(decimals_felt.try_into()?);
    }
}
```

Recommendation(s): Consider re-designing this function to return value directly instead of `Option`.

Status: Fixed.

Update from the client: Fix in audit_fixes branch: ab29258f27c338df29786f21f64e90612672fa11

6.25 [Best Practices] Unnecessary check of contract address and caller

File(s): `amm_core/liquidity_pool.cairo`

Description: Unnecessary check of contract address is zero. `get_contract_address(...)` method always returns a contract address and `get_caller_address(...)` method no longer returns zero.

```
fn deposit_liquidity(
        pooled_token_address: ContractAddress,
        quote_token_address: ContractAddress,
        base_token_address: ContractAddress,
        option_type: OptionType,
        amount: u256
) {

    // ...
    assert(!caller_addr.is_zero(), 'Caller address is zero');
    assert(!own_addr.is_zero(), 'Own address is zero'); // @audit-issue: Unnecessary checks
    // ...
 }
```

Recommendation(s): Consider removing these assertion statements.

Status: Fixed.

Update from the client: Fix in audit_fixed branch: bd13618572a512f885ca8d971721a91872ee72c0

6.26 [Best Practices] Unnecessary type casting

File(s): `amm_core/options.cairo`

Description: The function `_mint_option_token_long(...)` mints an option token long for the user. It modifies the long and short positions of the pool. The code snippet below shows the current long and short positions of the pool being fetched. Despite using the same function with the same return data type, `current_long_position` does not do type casting, while `current_short_position` does with the `into(...)` call. This type-casting is both unnecessary and inconsistent.

```
let current_long_position = get_option_position(
    lptoken_address, TRADE_SIDE_LONG, maturity, strike_price
);

let current_short_position = get_option_position(
    lptoken_address, TRADE_SIDE_SHORT, maturity, strike_price
)
    .into();
```

Recommendation(s): Consider removing the unnecessary `into(...)` call when getting the `current_short_position` value.

Status: Fixed.

Update from the client: Fix in audit_fixed branch: cf49021c79002b98dd864dc0a3b3fd481924ae87

6.27 [Best Practices] Unnecessary write to storage during migration

File(s): amm_core/state.cairo

Description: In the state.cairo contract, each getter first checks and migrates old data, then sets the old value to zero. However, this is unnecessary because the setters have already completed this task. For example, the get_option_volatility(...) function migrates the old value by calling set_option_volatility(...), then manually sets pool_volatility_separate to zero. This is redundant because set_option_volatility(...) has already performed the same operation.

```
fn get_option_volatility(
    lptoken_address: LPTAddress, maturity: Timestamp, strike_price: Strike
) -> Volatility {
    ...
    if res != 0 {
        // First assert it's not negative
        assert(res.into() > 0_u256, 'Old opt vol adj spd negative');

        // If it's not zero then move the old value to new storage var and set the old one to zero
        let res_cubit = FixedHelpersTrait::from_legacyMath(res);

        // Write old value to new storage var
        set_option_volatility(lptoken_address, maturity, strike_price, res_cubit);

        // Set old value to zero
        // @audit This has already been done in `set_option_volatility(...)`
        state
            .pool_volatility_separate
            .write((lptoken_address, maturity.into(), strike_price.to_legacyMath()), 0);

        return res_cubit;
}
```

Recommendation(s): Consider eliminating the unnecessary write to storage.

Status: Fixed.

Update from the client: Decided to remove the migrating functionality altogether: 636af75a91250f542582d413fd7cebf0b82381da

6.28 [Best Practices] Use standard traits for type conversions

File(s): types/option_.cairo

Description: Instead of writing your own function that returns the target type, use Into or TryInto trait implementations for your type conversions.

```
fn Option_to_LegacyOption(opt: Option_) -> LegacyOption {
    LegacyOption { // @audit-issue: Use Into or TryInto trait.
        option_side: opt.option_side,
        maturity: opt.maturity.into(),
        strike_price: opt.strike_price.to_legacyMath(),
        quote_token_address: opt.quote_token_address,
        base_token_address: opt.base_token_address,
        option_type: opt.option_type
    }
}

fn LegacyOption_to_Option(opt: LegacyOption) -> Option_ {
    Option_ { // @audit-issue: Use Into or TryInto trait.
        option_side: opt.option_side,
        maturity: opt.maturity.try_into().unwrap(),
        strike_price: FixedHelpersTrait::from_legacyMath(opt.strike_price),
        quote_token_address: opt.quote_token_address,
        base_token_address: opt.base_token_address,
        option_type: opt.option_type
    }
}
```

Recommendation(s): Consider implementing Into trait implementations between Option_ and LegacyOption types.

Status: Fixed.

Update from the client: Legacy types were removed since we're moving away from migration: 5ef83c22cadac0f5fd5a11422009be51cee83ab9

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. 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 Carmine documentation

The `Carmine` team has provided documentation on their protocol in the form of comments and their official documentation. Moreover, the Carmine team was available to address any inquiries or concerns raised by the Nethermind auditors.

8 Test Suite Evaluation

8.1 Compilation Output

```
scarb build
   Compiling carmine_protocol v0.1.0 (C:\Users\auditor\Desktop\Nethermind\protocol-cairo1\Scarb.toml)
    Finished release target(s) in 9 seconds
```

8.2 Tests Output

```
> snforge test
   Compiling carmine_protocol v0.1.0 (C:\Users\ermvr\OneDrive\Desktop\Nethermind\protocol-cairo1-master\Scarb.toml)
    Finished release target(s) in 9 seconds


Collected 81 test(s) from carmine_protocol package
Running 22 test(s) from src/
[PASS] carmine_protocol::amm_core::constants::tests::test_get_opposite_side
[PASS] carmine_protocol::amm_core::helpers::tests::test_get_underlying_from_option_data
[PASS] carmine_protocol::amm_core::helpers::tests::test_assert_option_side_exists
[PASS] carmine_protocol::amm_core::helpers::tests::test_assert_option_type_exists
[PASS] carmine_protocol::amm_core::helpers::tests::test__pow_failing
[PASS] carmine_protocol::amm_core::constants::tests::test_get_opposite_side_failing
[PASS] carmine_protocol::amm_core::helpers::tests::test_assert_option_type_exists_failing
[PASS] carmine_protocol::amm_core::helpers::tests::test_assert_option_side_exists_failing
[PASS] carmine_protocol::amm_core::helpers::tests::test_split_option_locked_capital
[PASS] carmine_protocol::amm_core::helpers::tests::test__pow
[PASS] carmine_protocol::amm_core::helpers::tests::test__toU256_balance
[PASS] carmine_protocol::amm_core::helpers::tests::test__fromU256_balance
[PASS] carmine_protocol::amm_core::pricing::option_pricing_helpers::tests::test_select_and_adjust_premia
[PASS] carmine_protocol::amm_core::pricing::option_pricing_helpers::tests::test_get_option_size_in_pool_currency
[PASS] carmine_protocol::amm_core::pricing::fees::tests::test_get_fees
[PASS] carmine_protocol::amm_core::pricing::option_pricing_helpers::tests::test_add_premia_fees
[PASS] carmine_protocol::amm_core::pricing::option_pricing::tests::test_black_scholes_extreme
[PASS] carmine_protocol::amm_core::pricing::option_pricing::tests::test_black_scholes
[PASS] carmine_protocol::amm_core::pricing::option_pricing_helpers::tests::test_get_time_till_maturity
[PASS] carmine_protocol::amm_core::pricing::option_pricing::tests::test_d1_d2
[PASS] carmine_protocol::amm_core::pricing::option_pricing_helpers::tests::test_get_new_volatility
[PASS] carmine_protocol::amm_core::pricing::option_pricing::tests::test_std_normal_cdf
Running 59 test(s) from tests/
[PASS] tests::test_sandwich_guard::test_sandwich_guard_mint_transfer_failing
[PASS] tests::test_sandwich_guard::test_sandwich_guard_mint_transfer
[PASS] tests::test_sandwich_guard::test_sandwich_guard_mint_transferFrom_failing
[PASS] tests::test_sandwich_guard::test_sandwich_guard_mint_mint
[PASS] tests::test_sandwich_guard::test_sandwich_guard_mint_burn_failing
[PASS] tests::test_sandwich_guard::test_sandwich_guard_mint_burn
[PASS] tests::test_sandwich_guard::test_sandwich_guard_mint_mint_failing
[PASS] tests::test_deposit_liquidity::test_deposit_liquidity_at_max_lpool_balance
[PASS] tests::test_expire_pool::test_expire_long
[PASS] tests::test_expire_pool::test_expire_short
[PASS] tests::test_dummy::test_dummy
[PASS] tests::test_sandwich_guard::test_sandwich_guard_burn_mint_failing
[PASS] tests::test_sandwich_guard::test_sandwich_guard_burn_transferFrom_failing
[PASS] tests::test_sandwich_guard::test_sandwich_guard_mint_transferFrom
[PASS] tests::test_sandwich_guard::test_sandwich_guard_burn_mint
[PASS] tests::test_sandwich_guard::test_sandwich_guard_burn_transfer_failing
[PASS] tests::test_sandwich_guard::test_sandwich_guard_burn_burn_failing
[PASS] tests::test_deposit_liquidity::test_deposit_liquidity
[PASS] tests::test_sandwich_guard::test_sandwich_guard_burn_transfer
[PASS] tests::test_sandwich_guard::test_sandwich_guard_burn_burn
[PASS] tests::test_sandwich_guard::test_sandwich_guard_burn_transferFrom
[PASS] tests::test_sandwich_guard::test_sandwich_guard_transfer_mint_same_block
[PASS] tests::test_sandwich_guard::test_sandwich_guard_transfer_burn_same_block
[PASS] tests::test_sandwich_guard::test_sandwich_guard_transfer_transfer_same_block
[PASS] tests::test_sandwich_guard::test_sandwich_guard_transfer_transferFrom_same_block
[PASS] tests::test_sandwich_guard::test_sandwich_guard_transfer_mint_next_block
[PASS] tests::test_sandwich_guard::test_sandwich_guard_transfer_burn_next_block
```

```
[PASS] tests::test_sandwich_guard::test_sandwich_guard_transfer_transfer_next_block
[PASS] tests::test_sandwich_guard::test_sandwich_guard_transfer_transferFrom_next_block
[PASS] tests::test_trade_close::test_trade_close_long
[PASS] tests::test_trade_close::test_trade_close_short
[PASS] tests::test_trade_close::test_trade_close_long_with_pool_position_change
[PASS] tests::test_trade_close::test_trade_close_short_with_pool_position_change
[PASS] tests::test_trade_close::test_trade_close_long_with_pool_long_position
[PASS] tests::test_trade_close::test_trade_close_long_with_pool_short_position
[PASS] tests::test_trade_open::test_trade_open_long
[PASS] tests::test_trade_open::test_trade_open_short
[PASS] tests::test_trade_close::test_trade_close_short_with_pool_long_position
[PASS] tests::test_trade_open::test_trade_open_long_pool_long_position
[PASS] tests::test_trade_close::test_trade_close_short_with_pool_short_position
[PASS] tests::test_trade_open::test_trade_open_short_pool_short_position
[PASS] tests::test_trading_halt::test_set_trading_halt_permission
[PASS] tests::test_trade_open::test_trade_open_long_with_pool_position_change
[PASS] tests::test_trading_halt::test_set_trading_halt_permission_failing
[PASS] tests::test_trade_settle::test_trade_settle_long
[PASS] tests::test_trade_open::test_trade_open_short_with_pool_position_change
[PASS] tests::test_trade_settle::test_trade_settle_short
[PASS] tests::test_trading_halt::test_set_trading_halt
[PASS] tests::test_trading_halt::test_set_trading_halt_failing
[PASS] tests::test_trading_halt::test_set_trading_halt_trade_failing
[PASS] tests::test_view::test_get_all_lptoken_addresses
[PASS] tests::test_view::test_get_all_options
[PASS] tests::test_view::test_get_all_poolinfo
[PASS] tests::test_view::test_get_total_premia
[PASS] tests::test_view::test_get_user_pool_infos
[PASS] tests::test_withdraw_liquidity::test_withdraw_liquidity_not_enough_capital
[PASS] tests::test_view::test_get_all_non_expired_options_with_premia
[PASS] tests::test_withdraw_liquidity::test_withdraw_liquidity_not_enough_lptokens
[PASS] tests::test_withdraw_liquidity::test_withdraw_liquidity
Tests: 81 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