CAIRO SECURITY CLAN

AVNU - DCA

SECURITY ASSESEMENT REPORT

JULY 2024

Prepared for

AVNU

Contents

1 About Cairo Security Clan

Cairo Security Clan is a leading force in the realm of blockchain security, dedicated to fortifying the foundations of the digital age. As pioneers in the field, we specialize in conducting meticulous smart contract security audits, ensuring the integrity and reliability of decentralized applications built on blockchain technology.

At Cairo Security Clan, we boast a multidisciplinary team of seasoned professionals proficient in blockchain security, cryptography, and software engineering. With a firm commitment to excellence, our experts delve into every aspect of the Web3 ecosystem, from foundational layer protocols to application-layer development. Our comprehensive suite of services encompasses smart contract audits, formal verification, and real-time monitoring, offering unparalleled protection against potential vulnerabilities.

Our team comprises industry veterans and scholars with extensive academic backgrounds and practical experience. Armed with advanced methodologies and cutting-edge tools, we scrutinize and analyze complex smart contracts with precision and rigor. Our track record speaks volumes, with a plethora of published research papers and citations, demonstrating our unwavering dedication to advancing the field of blockchain security.

At Cairo Security Clan, we prioritize collaboration and transparency, fostering meaningful partnerships with our clients. We believe in a customer-oriented approach, engaging stakeholders at every stage of the auditing process. By maintaining open lines of communication and soliciting client feedback, we ensure that our solutions are tailored to meet the unique needs and objectives of each project.

Beyond our core services, Cairo Security Clan is committed to driving innovation and shaping the future of blockchain technology. As active contributors to the ecosystem, we participate in the development of emerging technologies such as Starknet, leveraging our expertise to build robust infrastructure and tools. Through strategic guidance and support, we empower our partners to navigate the complexities of the blockchain landscape with confidence and clarity.

In summary, Cairo Security Clan stands at the forefront of blockchain security, blending technical prowess with a client-centric ethos to deliver unparalleled protection and peace of mind in an ever-evolving digital landscape. Join us in safeguarding the future of decentralized finance and digital assets with confidence and conviction.

2 Disclaimer

Disclaimer Limitations of this Audit:

This report is based solely on the materials and documentation provided by you to Cairo Security Clan for the specific purpose of conducting the security review outlined in the Summary of Audit and Scoped Files. The findings presented here may not be exhaustive and may not identify all potential vulnerabilities. Cairo Security Clan provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, occurs entirely at your own risk.

Inherent Risks of Blockchain Technology:

Blockchain technology remains in its developmental stage and is inherently susceptible to unknown risks and vulnerabilities. This review is specifically focused on the smart contract code and does not extend to the compiler layer, programming language elements beyond the reviewed code, or other potential security risks outside the code itself.

Report Purpose and Reliance:

This report should not be construed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. No third party should rely on this report for any purpose, including making investment or purchasing decisions.

Liability Disclaimer:

To the fullest extent permitted by law, Cairo Security Clan disclaims all liability associated with this report, its contents, and any related services and products arising from your use. This includes, but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services:

Cairo Security Clan does not warrant, endorse, guarantee, or assume responsibility for any products or services advertised by third parties within this report, nor for any open-source or third-party software, code, libraries, materials, or information linked to, referenced by, or accessible through this report, its content, and related services and products. This includes any hyperlinked websites, websites or applications appearing on advertisements, and Cairo Security Clan will not be responsible for monitoring any transactions between you and third-party providers. It is recommended that you exercise due diligence and caution when considering any third-party products or services, just as you would with any purchase or service through any medium.

Disclaimer of Advice:

FOR THE AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, ACCESS, AND/OR USE, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

3 Executive Summary

This document presents the security review performed by Cairo Security Clan on the Avnu.

AVNU is a decentralized exchange protocol designed to offer the best execution in DeFi for Layer 2. AVNU's primary goal is to provide better pricing, zero slippage, MEV-protection, and gasless trading, ensuring the most efficient and seamless operations in the DeFi space.Learn more from docs.

The audit was performed using

- manual analysis of the codebase,
- automated analysis tools,
- simulation of the smart contract,
- analysis of edge test cases

6 points of attention, where 0 is classified as Critical, 0 is classified as High, 2 are classified as Medium,3 are classified as Low,1 is classified as Informational and 0 is classified as Best Practices. The issues are summarized in Fig. 1.

This document is organized as follows. Section 1 About Cairo Security Clan. Section 2 Disclaimer. Section 3 Executive Summary. Section 4 Summary of Audit. Section 5 Risk Classification. Section 6 Issues by Severity Levels. Section 7 Test Evaluation.

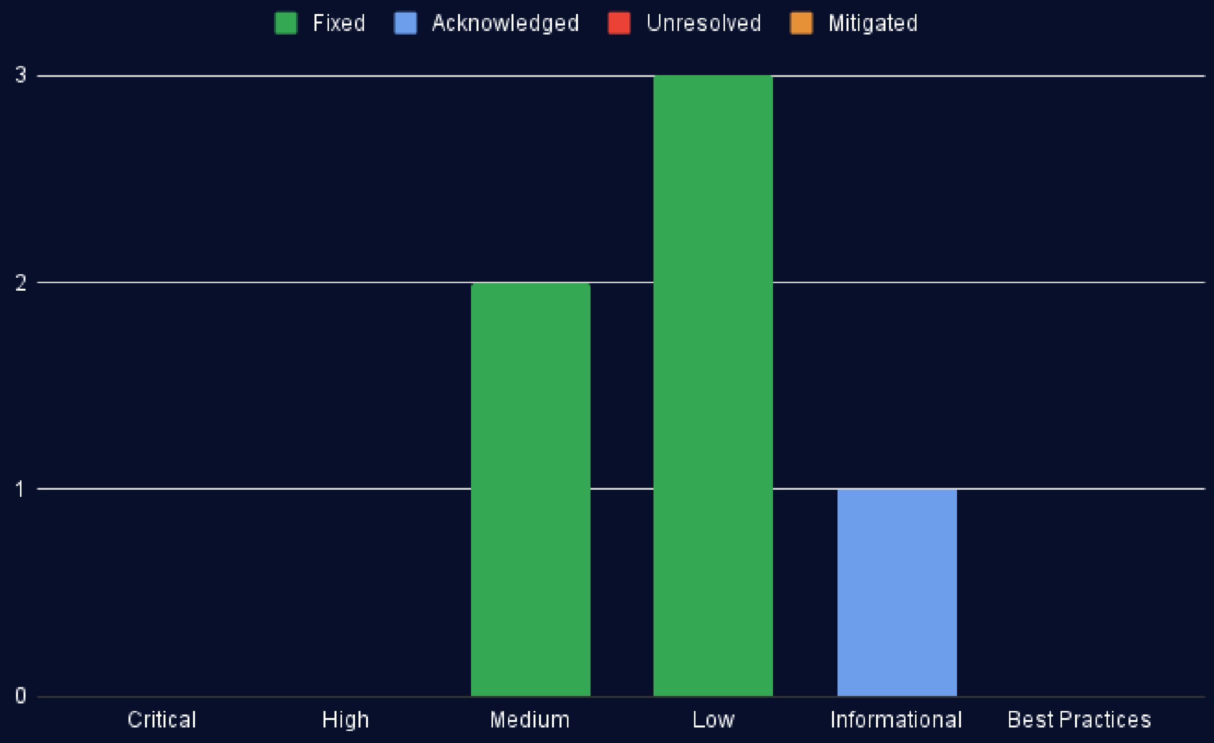

Fig 1: Distribution of issues: Critical (0), **High** (0), **Medium** (2), **Low** (3), **Informational** (1), **Best Practices** (0).
Distribution of status: Fixed (5), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0).

4 Summary of Audit

Audit Type	Security Review
Cairo Version	2.6.3
Response from Client	25/07/2024
Final Report	30/07/2024
Repository	avnu-labs/dca-contracts
Initial Commit Hash	3e3d4d10b59f13d3d46b68bb9d086b48021f5774
Final Commit Hash	572237a7bd0814d59783c8b35c7096f56f711778
Documentation	Website documentation
Test Suite Assessment	High

4.1 Scoped Files

	Contracts
1	src/components.cairo
2	src/lib.cairo
3	src/orchestrator.cairo
4	src/order.cairo
5	src/components/fee.cairo
6	src/components/ownable.cairo

4.2 Issues

	Findings	**Severity**	**Update**
1	Arbitrary call allows whitelisted caller to steal from any trader	Medium	Fixed
2	The total swapped amount could be larger than the `token_from_amount` specified by traders	Medium	Fixed
3	Whitelisted callers can input arbitrary trader address to receive the swapped token	Low	Fixed
4	Function `execute_batch()` does not validate token address match with trader's orders	Low	Fixed
5	Trader's order will have one more cycle than expected	Low	Fixed
6	Unchecked return values of ERC20 transfer	Informational	Acknowledged

5 Risk Classification

The risk rating methodology used by Cairo Security Clan follows the principles established by the CVSS risk rating methodology. 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.

		Likelihood		
		High	**Medium**	**Low**
Impact	**High**	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Info/Best Practices

To address issues that do not fit a High/Medium/Low severity, Cairo Security Clan also uses three more finding severities: **Informational**, **Best Practices** and **Gas**

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;

b) **Gas** findings are used when some piece of code uses more gas than it should be or have some functions that can be removed to save gas.

6 Issues by Severity Levels

6.1 Medium

6.1.1 Arbitrary call allows whitelisted caller to steal from any trader

File(s): src/orchestrator.cairo

Description: In the orchestrator contract, the entry point function `execute_batch()` can only be called by a whitelisted caller address. The list of traders is passed to the function, the function loops through each trader and calls the function `initiate_order_execution()` to retrieve the input token from traders. Then, it executes a list of arbitrary calls to perform the swap.

```
let traders = self.retrieve_token_from(contract_address, token_from, traders.span());
let token_from_amount = token_from.balanceOf(contract_address);

// Execute Swap
while let Option::Some(call) = calls
    .pop_front() {
        call_contract_syscall(call.to, call.selector, call.calldata).unwrap_syscall();
    };
```

This design is risky and assumes that users trust the whitelisted callers. If the whitelisted callers act maliciously, they can steal funds from any trader by including these two calls in the `calls` list:

– The first call is `retrieve_token_from()` to retrieve funds from traders through the order contract.

– The second call is `token_from.transfer()` to transfer/steal all retrieved funds.

Recommendation(s): Consider whitelisting the address and selector that the contract is able to call.

Status: Fixed

Update from client: Fixed in this commit.

6.1.2 The total swapped amount could be larger than the `token_from_amount` specified by traders

File(s): src/order.cairo

Description: In each DCA order, the trader can specify the total amount of input token through `token_from_amount` and the amount to swap per cycle through `token_from_amount_per_cycle`. The number of cycles needed to swap all of them would be rounded up if `token_from_amount` is not divisible by `token_from_amount_per_cycle`. In this case, the last cycle would need to swap less than a normal cycle. For example, if `token_from_amount` = 160 and `token_from_amount_per_cycle` = 50, we need to swap in 3 normal cycles, and the 4th cycle would only swap 10 tokens.

However, the function `initiate_order_execution()` does not have that logic and always tries to swap `token_from_amount_per_cycle`. As a result, the total swapped amount could be larger than the `token_from_amount` specified by the trader. In the example above, the total swapped amount would be 200, while the trader only wants to swap 160.

```
fn initiate_order_execution(ref self: ContractState) -> u256 {
    //...

    // Transfer tokens to DCA Orchestrator
    token_from.transferFrom(trader_address, contract_address_const::<ORCHESTRATOR_ADDRESS>(), order.
     token_from_amount_per_cycle);
    order.token_from_amount_used += order.token_from_amount_per_cycle;

    self.order.write(order);
    order.token_from_amount_per_cycle
}
```

Recommendation(s): Consider adding a check in the constructor to ensure `token_from_amount` is divisible by `token_from_amount_per_cycle`.

Status: Fixed

Update from client: Fixed in this commit.

6.2 Low

6.2.1 Whitelisted callers can input arbitrary trader address to receive the swapped token

File(s): src/orchestrator.cairo

Description: When the caller invokes the `execute_batch()` function to execute a batch trade, the list of traders is passed to the function. This function then loops through each trader in the list to retrieve funds from them. After performing the swap, it loops through the list again to send the swapped token to each `trader.address`.

```
while let Option::Some(trader) = traders
  .pop_front() {
      let amount_to_send = if (traders.len() == 0) {
          ...
      };
      IDcaOrderDispatcher { contract_address: *trader.order_address }.fulfill_order_execution(amount_to_send);
      token_to_amount_sent += amount_to_send;

      token_to.transfer(*trader.address, amount_to_send);
      traders_with_token_to_amount
          .append(
              ...
          );
  };
```

However, there is no validation to ensure that the `trader.address` is the owner of the order contract. Callers can input an arbitrary address to receive the funds after swapping, effectively stealing from traders.

Recommendation(s): Consider validating that `trader.address` is the same as the owner of the order contract.

Status: Fixed

Update from client: Fixed in this commit.

6.2.2 Function `execute_batch()` does not validate token address match with trader's orders

File(s): src/orchestrator.cairo

Description: When the caller invokes the `execute_batch()` function to execute a batch trade, the token from and token to addresses are passed by the caller to the function. This function then loops through each trader in the traders list to retrieve funds from the traders.

However, there is no check to ensure the token addresses in the trader's orders match the addresses passed by the caller. As a result, the caller might pass in invalid traders (with different token addresses), and the function will not revert. The incorrect tokens will just be stuck in the contract.

```
fn retrieve_token_from(
    ref self: ContractState, contract_address: ContractAddress, token_from: IERC20Dispatcher, mut traders: Span<
     Trader>
) -> Array<TraderWithTokenFromAmount> {
    let mut traders_with_token_from_amount: Array<TraderWithTokenFromAmount> = array![];

    while let Option::Some(trader) = traders
        .pop_front() {
            let token_from_amount = IDcaOrderDispatcher { contract_address: *trader.order_address }.
     initiate_order_execution();
            traders_with_token_from_amount
                .append(
                    TraderWithTokenFromAmount { address: *trader.address, order_address: *trader.order_address,
     token_from_amount }
                )
        };
    traders_with_token_from_amount
}
```

Recommendation(s): Consider adding a check to ensure the token addresses in the trader's orders match the addresses passed into the function.

Status: Fixed

Update from client: Fixed in this commit.

6.2.3 Trader's order will have one more cycle than expected

File(s): src/order.cairo

Description: In each DCA order, the trader can specify the total amount of the input token through `token_from_amount` and the amount to swap per cycle through `token_from_amount_per_cycle`. The constructor uses these two values to calculate the `iterations` value, which is the number of cycles the trade needs to be executed. The `end_date` is then calculated by adding the result of multiplying `cycle_frequency` and iterations with `next_cycle_at`.

```
let (iterations, _, _) = u256_safe_divmod(token_from_amount, u256_as_non_zero(token_from_amount_per_cycle));
...
let end_date = next_cycle_at + cycle_frequency * iterations.try_into().unwrap();
```

However, the function `assert_next_trade_available()` checks the order end date using the condition `block_timestamp <= *order.end_-date`. As a result, the order could be executed for one more cycles than it should be. Consider the scenario below:

– Alice (a trader) opens an order with the following parameters

```
token_from_amount = 2000
token_from_amount_per_cycle = 500
cycle_frequency = 3600 (1 hour)
start_at = 0 (assuming block_timestamp = 0)
```

– The calculation in the order contract constructor will be:

```
iterations = token_from_amount / token_from_amount_per_cycle = 4
next_cycle_at = 0
end_date = next_cycle_at + 3600 * 4 = 14400
```

As we can see, even though there should be only 4 cycles, the trade could be executed at 5 timestamps (5 cycles): 0, 3600, 7200, 10800, 14400.

Recommendation(s): Consider changing the way `end_date` is calculated in the constructor or not allowing the trade to executed at `end_date`.

Status: Fixed

Update from client: Fixed in this commit.

6.3 Informationals

6.3.1 Unchecked return values of ERC20 transfer

File(s): src/orchestrator.cairo , src/order.cairo

Description: In the codebase, there are two ERC20 transfers to move tokens between the trader's wallet and the orchestrator contract using `transfer()` and `transferFrom()`. These functions return a boolean value indicating whether the transfer succeeded or not. However, these return values are not currently checked, potentially resulting in unexpected behavior, especially if the token does not revert on failure.

```
/// orchestrator.cairo
token_to.transfer(*trader.address, amount_to_send);

/// order.cairo
// Transfer tokens to DCA Orchestrator
token_from.transferFrom(trader_address, contract_address_const::<ORCHESTRATOR_ADDRESS>(), order.
    token_from_amount_per_cycle);
```

Recommendation(s): Consider adding a check for the boolean return values.

Status: Acknowledged

Update from client: We don't think that this is necessary. If the ERC20 transfer fails, the transaction will be rejected (or reverted).

7 Test Evaluation

7.1 Compilation Output

```
Run scarb build
    Updating git repository github.com/avnu-labs/avnu-contracts-lib
    Updating git repository github.com/openzeppelin/cairo-contracts
    Updating git repository github.com/foundry-rs/starknet-foundry
   Compiling avnu_dca v0.1.0 (/014-AVNU-DCA/014-AVNU-DCA/contracts/Scarb.toml)
    Finished release target(s) in 4 s
```

7.2 Tests Output

7.2.1 Cairo Tests

```
Run scarb test
     Running test avnu_dca (sed -i s/0x0492139c56af6faf77119b6bca3b6d40f559af6b7b23778f068dd9ca08e407c5
     /1166415730164802550690328744940341223797107815383083168824998551848908409526/g src/order.cairo && scarb
     cairo-test && sed -i s/1166415730164802550690328744940341223797107815383083168824998551848908409526/0
     x0492139c56af6faf77119b6bca3b6d40f559af6b7b23778f068dd9ca08e407c5/g src/order.cairo)
   Compiling test(avnu_dca_unittest) avnu_dca v0.1.0 (/014-AVNU-DCA/014-AVNU-DCA/contracts/Scarb.toml)
   Compiling test(avnu_dca_tests) avnu_dca_tests v0.1.0 (/014-AVNU-DCA/014-AVNU-DCA/contracts/Scarb.toml)
    Finished release target(s) in 25 s
testing avnu_dca ...
running 37 tests
test avnu_dca_tests::components::fee_test::GetFessRecipient::should_return_fees_recipient ... ok (gas usage est.:
      308220)
test avnu_dca_tests::order_test::Constructor::should_fail_when_token_from_amount_per_cycle_is_zero ... ok (gas
     usage est.: 1298320)
test avnu_dca_tests::components::fee_test::SetFessRecipient::should_set_fess_recipient ... ok (gas usage est.:
     435490)
test avnu_dca_tests::components::fee_test::SetFessRecipient::should_fail_when_caller_is_not_the_owner ... ok (gas
      usage est.: 334430)
test avnu_dca_tests::order_test::Constructor::
     should_fail_when_token_from_amount_per_cycle_higher_than_token_from_amount ... ok (gas usage est.: 1298320)
test avnu_dca_tests::components::fee_test::GetFeesBps::should_return_bps ... ok (gas usage est.: 308620)
test avnu_dca_tests::order_test::Constructor::should_fail_when_too_many_iterations ... ok (gas usage est.:
     1298320)
test avnu_dca_tests::components::fee_test::SetFeesBps0::should_set_fees_bps_0 ... ok (gas usage est.: 437820)
test avnu_dca_tests::components::fee_test::SetFeesBps0::should_fail_when_fees_are_too_high ... ok (gas usage est
     .: 336360)
test avnu_dca_tests::order_test::Constructor::should_fail_when_not_enough_iterations ... ok (gas usage est.:
     1298320)
test avnu_dca_tests::components::fee_test::SetFeesBps0::should_fail_when_caller_is_not_the_owner ... ok (gas
     usage est.: 336360)
test avnu_dca_tests::order_test::Constructor::should_fail_when_cycle_frequency_is_too_low ... ok (gas usage est.:
      1297750)
test avnu_dca_tests::components::ownable_test::GetOwner::should_return_owner ... ok (gas usage est.: 272040)
test avnu_dca_tests::order_test::Constructor::should_fail_when_cycle_frequency_is_too_high ... ok (gas usage est
     .: 1299260)
test avnu_dca_tests::order_test::Constructor::should_fail_when_allows_is_too_low ... ok (gas usage est.: 1104860)
test avnu_dca_tests::order_test::IsNextTradeAvailable::should_return_true_when_next_trade_is_available ... ok (
     gas usage est.: 2281430)
test avnu_dca_tests::order_test::IsNextTradeAvailable::should_fail_when_order_is_not_open ... ok (gas usage est.:
      3098590)
test avnu_dca_tests::order_test::IsNextTradeAvailable::should_fail_when_next_cycle_not_available ... ok (gas
     usage est.: 2080620)
test avnu_dca_tests::order_test::IsNextTradeAvailable::should_fail_when_date_is_after_end_date ... ok (gas usage
     est.: 2061990)
test avnu_dca_tests::order_test::CloseOrder::should_close_order ... ok (gas usage est.: 3060540)
test avnu_dca_tests::order_test::CloseOrder::should_fail_when_caller_is_not_the_owner ... ok (gas usage est.:
     2229030)
test avnu_dca_tests::orchestrator_test::Execute::should_execute_a_dca_batch_with_multiple_traders ... ok (gas
     usage est.: 18274150)
test avnu_dca_tests::order_test::CloseOrder::should_fail_when_allowance_is_not_zero ... ok (gas usage est.:
     2415340)
```

```
test avnu_dca_tests::order_test::FulfillOrderExecution::should_check_pricing_strategy ... ok (gas usage est.:
     2071130)
test avnu_dca_tests::order_test::FulfillOrderExecution::should_fail_when_token_to_amount_is_too_low ... ok (gas
     usage est.: 2070230)
test avnu_dca_tests::order_test::FulfillOrderExecution::should_fail_when_token_to_amount_is_too_high ... ok (gas
     usage est.: 2070230)
test avnu_dca_tests::orchestrator_test::Execute::should_execute_a_dca_batch ... ok (gas usage est.: 7120200)
test avnu_dca_tests::order_test::FulfillOrderExecution::should_fail_when_caller_is_not_the_orchestrator ... ok (
     gas usage est.: 1980230)
test avnu_dca_tests::order_test::InitiateOrderExecution::should_close_order_when_execute_latest_iteration ... ok
     (gas usage est.: 4775790)
test avnu_dca_tests::orchestrator_test::Execute::should_fail_when_token_from_address_is_0 ... ok (gas usage est.:
      2496910)
test avnu_dca_tests::orchestrator_test::Execute::
     should_execute_a_dca_batch_and_send_remaining_token_from_to_fee_recipient ... ok (gas usage est.: 7653860)
test avnu_dca_tests::orchestrator_test::Execute::should_fail_when_token_to_address_is_0 ... ok (gas usage est.:
     2496910)
test avnu_dca_tests::orchestrator_test::Execute::should_fail_when_traders_is_empty ... ok (gas usage est.:
     1403470)
test avnu_dca_tests::orchestrator_test::Execute::should_fail_when_token_to_amount_is_not_sufficient ... ok (gas
     usage est.: 5243270)
test avnu_dca_tests::orchestrator_test::Execute::should_fail_when_calls_is_empty ... ok (gas usage est.: 2373480)
test avnu_dca_tests::orchestrator_test::Execute::should_fail_when_caller_is_not_whitelisted ... ok (gas usage est
     .: 2367900)
test avnu_dca_tests::order_test::Constructor::should_init_order ... ok (gas usage est.: 3744340)

test result: ok. 37 passed; 0 failed; 0 ignored; 0 filtered out;
```