Security Review Report
NM-0258 INFLUENCETH SWAY CONTRACT

NETHERMIND
SECURITY

(Jun 21, 2024)

Contents

1 Executive Summary **2**

2 Audited Files **3**

3 Summary of Issues **3**

4 System Overview **4**
4.1 Contract Storage . . . 5
4.2 Events . . . 5

5 Risk Rating Methodology **6**

6 Issues **7**
6.1 [Info] Centralization risks . . . 7
6.2 [Info] Consider emitting when updating the end-points of the bridge . . . 7
6.3 [Info] Unused storage variable `l1_sway_address` . . . 7
6.4 [Info] Using zero value as `memo` can lead to transfers without confirmation . . . 8
6.5 [Info] Volume of transactions can be artificially inflated . . . 8

7 Documentation Evaluation **9**

8 Test Suite Evaluation **10**
8.1 Complementary Tests . . . 13

9 About Nethermind **16**

1 Executive Summary

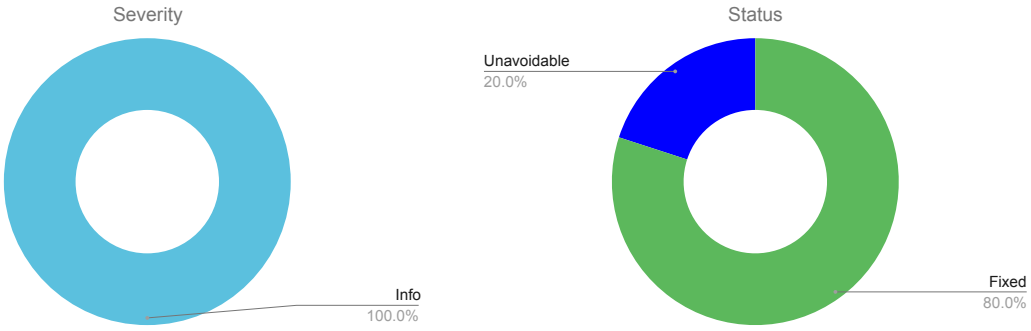
This document outlines the security review conducted by Nethermind Security for the INFLUENCETH SWAY contract. SWAY is an ERC-20 contract written in Cairo language. Apart from the standard ERC-20 operations, the contract also implements:

- A mechanism to keep the history of the transfers during a given period of time. In the audited version, this value corresponds to 1,000,000 (one million seconds, i.e., something around 11.5 days);
- Access Control allowing for minting and setting operational parameters;
- Logic for bridging assets between Layers 1 and 2;
- This version can update the contract.

The audited code comprises 648 lines of code in Cairo. The **INFLUENCETH** team has provided documentation explaining the audited contracts' purpose and functionality. The audit was conducted using: (a) manual analysis of the codebase; (b) simulation of the smart contracts; and (c) evaluation of the test suite along with the creation of new test cases.

Throughout this document, we highlight five points of attention, classified as Info. Four out of these five points have been fixed by the SWAY team. The remaining issue pertains to the risk of malicious users artificially inflating the volume of transfers per period, which cannot be mitigated. 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 discusses the 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 (0), **High** (0), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (5), **Best Practices** (0). **(b) Distribution of status: Fixed** (4), **Unavoidable** (1), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jun 19, 2024
Final Report	Jun 20, 2024
Methods	Manual Review, Automated analysis, Creation of test cases
Repository	SWAY
Commit Hash	d6d5262dcf56b5db46d5cfa0ea5f102571d9d8c7
Final Commit Hash	cd9a523295ec3d8ea2c48672af2a0a166791e364
Documentation	INFLUENCETH wiki
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/contracts/sway.cairo	633	127	20.1%	168	928
	Total	**633**	**127**	**20.1%**	**168**	**928**

3 Summary of Issues

	Finding	Severity	Update
1	Centralization risks	Info	Fixed
2	Consider emitting when updating the end-points of the bridge	Info	Fixed
3	Unused storage variable `l1_sway_address`	Info	Fixed
4	Using zero value as `memo` can lead to transfers without confirmation	Info	Fixed
5	Volume of transactions can be artificially inflated	Info	Unavoidable

4 System Overview

SWAY implements an ERC-20 contract plus bridging functionalities and a mechanism to keep track of the total value transfered in periods of 1,000,000 seconds (something around 11.5 days). The interface for the ERC-20 functions are reproduced below.

```
fn allowance(self: @TContractState, owner: ContractAddress, spender: ContractAddress) -> u256;
fn approve(ref self: TContractState, spender: ContractAddress, amount: u256) -> bool;
fn balance_of(self: @TContractState, account: ContractAddress) -> u256;
fn balanceOf(self: @TContractState, account: ContractAddress) -> u256;
fn decimals(self: @TContractState) -> u8;
fn decrease_allowance(ref self: TContractState, spender: ContractAddress, subtracted_value: u256);
fn decreaseAllowance(ref self: TContractState, spender: ContractAddress, subtracted_value: u256);
fn increase_allowance(ref self: TContractState, spender: ContractAddress, added_value: u256);
fn increaseAllowance(ref self: TContractState, spender: ContractAddress, added_value: u256);
fn name(self: @TContractState) -> felt252;
fn symbol(self: @TContractState) -> felt252;
fn total_supply(self: @TContractState) -> u256;
fn totalSupply(self: @TContractState) -> u256;
fn transfer(ref self: TContractState, recipient: ContractAddress, amount: u256) -> bool;
fn transfer_from(ref self: TContractState, sender: ContractAddress, recipient: ContractAddress, amount: u256) -> bool;
fn transferFrom(ref self: TContractState, sender: ContractAddress, recipient: ContractAddress, amount: u256) -> bool;
fn mint(ref self: TContractState, recipient: ContractAddress, amount: u256);
}
```

The contract also implements special `transfer` functions with a confirmation scheme. The header of these functions is shown below.

```
fn transfer_with_confirmation(
    ref self: TContractState,
    recipient: ContractAddress,
    amount: u128,
    memo: felt252,
    consumer: ContractAddress
) -> bool;

fn transfer_from_with_confirmation(
    ref self: TContractState,
    sender: ContractAddress,
    recipient: ContractAddress,
    amount: u128,
    memo: felt252,
    consumer: ContractAddress
) -> bool;

fn confirm_receipt(
    ref self: TContractState, sender: ContractAddress, recipient: ContractAddress, amount: u128, memo: felt252
);
```

The contract also implements access control via the functions shown below.

```
fn add_grant(ref self: TContractState, account: ContractAddress, role: u64);
fn has_grant(self: @TContractState, account: ContractAddress, role: u64) -> bool;
fn revoke_grant(ref self: TContractState, account: ContractAddress, role: u64);
```

Finally, we have the code responsible for bridging assets between the layers.

```
fn get_l1_bridge(self: @TContractState) -> EthAddress;
fn get_l2_token(self: @TContractState) -> ContractAddress;
fn set_l1_bridge(ref self: TContractState, l1_bridge: EthAddress);
fn initiate_withdrawal(ref self: TContractState, l1_recipient: EthAddress, amount: u256);
fn handle_deposit(ref self: TContractState,
    from_address: felt252, // L1 Bridge
    to_address: ContractAddress, // L2 destination account
    amount_low: felt252, amount_high: felt252 );

fn set_l1_sway_address(ref self: TContractState, l1_sway_address: EthAddress);
fn set_l1_sway_volume_address(ref self: TContractState, l1_sway_volume_address: EthAddress);
```

4.1 Contract Storage

The contract storage is composed of the following fields.

```
#[storage]
struct Storage {
    _decimals: u8,
    _name: felt252,
    _symbol: felt252,
    _total_supply: u256,
    allowances: LegacyMap::<(ContractAddress, ContractAddress), u256>,
    balances: LegacyMap::<ContractAddress, u256>,
    confirmations: LegacyMap::<felt252, felt252>,
    l1_bridge_address: EthAddress,
    l1_sway_volume_address: EthAddress,
    period_volumes: LegacyMap::<u64, u256>,
    role_grants: LegacyMap::<(ContractAddress, u64), bool>,
}
```

We can notice that standard elements of an `ERC-20` contract, such as `_decimals`, `_name`, `_symbol`, `_total_supply`, `allowances mapping`, and the `balances mapping`.

We also notice fields required to implement the complementary functionalities:

```
confirmations: LegacyMap::<felt252, felt252>,         // implements the mapping used for transfer confirmations;
l1_bridge_address: EthAddress,                         // stores the address of the L1 bridge;
l1_sway_address: EthAddress,
l1_sway_volume_address: EthAddress,                    // address of the swai contract in Layer 1;
period_volumes: LegacyMap::<u64, u256>,               // stores the volume of transfers of each 1_000_000 seconds window;
role_grants: LegacyMap::<(ContractAddress, u64), bool>, // mapping for access control
```

4.2 Events

The contract also emits the following events.

```
enum Event {
    Transfer: Transfer,
    Approval: Approval,
    DepositHandled: DepositHandled,
    WithdrawInitiated: WithdrawInitiated,
    ConfirmationCreated: ConfirmationCreated,
    ReceiptConfirmed: ReceiptConfirmed
}
```

We recommend the contract to also emit events when updating the end-points of the bridge.

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 [Info] Centralization risks

File(s): src/contracts/sway.cairo

Description: Centralization risks in smart contracts refer to drawbacks associated with a high degree of control or influence held by a single entity or a small group of entities within the smart contract. The contract has an administrator who can execute special actions without the consent of users. For instance, the administrator can `mint`, upgrade the contract, change the bridge address, and add/remove new administrators.

Recommendation(s): We suggest updating the user-facing documentation to explain the situations when the administrator can take these special actions.

Status: Fixed

Update from the client: Notes have been added to make clear that the initial admin role will be revoked after the audit is complete.

6.2 [Info] Consider emitting when updating the end-points of the bridge

File(s): src/contracts/sway.cairo

Description: The administrator (`ADMIN`) can update the address of the `L1` bridge. This operation is critical, and it would be advantageous for the contract to emit an event each time the bridge address is updated. The code, including audit comments, is provided below.

```
#[external(v0)]
fn set_l1_bridge(ref self: ContractState, l1_bridge: EthAddress) {
    ////////////////////////////////////
    // @audit: consider emiting here
    ////////////////////////////////////
    assert(self.role_grants.read((get_caller_address(), roles::ADMIN)), 'SWAY: must be admin');
    self.l1_bridge_address.write(l1_bridge);
}
```

Recommendation(s): Consider emiting an event whenever the bridge address is updated.

Status: Fixed

Update from the client: An emitted event has been added:

```
#[derive(Drop, starknet::Event)]
struct L1BridgeUpdated {
    address: EthAddress
}

#[external(v0)]
fn set_l1_bridge(ref self: ContractState, l1_bridge: EthAddress) {
    assert(self.role_grants.read((get_caller_address(), roles::ADMIN)), 'SWAY: must be admin');
    self.l1_bridge_address.write(l1_bridge);
    self.emit(L1BridgeUpdated { address: l1_bridge });
}
```

6.3 [Info] Unused storage variable `l1_sway_address`

File(s): src/contracts/sway.cairo

Description: The storage variable `l1_sway_address` is never used.

Recommendation(s): Remove the unused variable.

Status: Fixed

Update from the client: This variable was deprecated once the determination was made not to use a monolithic token / bridge on L1, it has been removed along with its setter.

6.4 [Info] Using zero value as `memo` can lead to transfers without confirmation

File(s): src/contracts/sway.cairo

Description: The `transfer_with_confirmation(...)` function does not ensure that `memo` is non-zero. This oversight allows users to make transfers through `transfer_with_confirmation(...)` without effectively creating a confirmation. The function is shown below.

```
fn transfer_with_confirmation(
        ref self: ContractState,
        recipient: ContractAddress,
        amount: u128,
        memo: felt252,
        consumer: ContractAddress
    ) -> bool {
        let caller_address: ContractAddress = starknet::get_caller_address();

        let mut to_hash: Array<felt252> = Default::default();
        to_hash.append(caller_address.into());
        to_hash.append(recipient.into());
        to_hash.append(amount.into());
        ////////////////////////////////////////////////
        // @audit: missing validation for field "memo"
        ////////////////////////////////////////////////
        to_hash.append(memo);
        to_hash.append(consumer.into());
        let hashed = poseidon::poseidon_hash_span(to_hash.span());

        assert(self.confirmations.read(hashed) == 0, 'SWAY: already confirmed');
        self.confirmations.write(hashed, memo);
        transfer(ref self, recipient, amount.into());

        self.emit(ConfirmationCreated {
            from: caller_address,
            to: recipient,
            value: amount,
            memo: memo,
            consumer: consumer
        });

        return true;
    }
```

Recommendation(s): Ensure that `memo` is different from zero.

Status: Fixed

Update from the client: We don't think it's a problem, as, in general, it's in the sending account's best interest to include the correct memo (since the receipt is confirmed by the game contract in their favor), but we agree it's unexpected behavior. We have added an assertion that the memo is present.

6.5 [Info] Volume of transactions can be artificially inflated

File(s): src/contracts/sway.cairo

Description: Sway tracks the amount transferred in each 1-million-second period. Malicious users can transfer funds between their accounts to artificially inflate the transaction volume during a given period.

Recommendation(s): This cannot be prevented. Users can always create more accounts and transfer funds between them. This is intrinsic to any system that allows transfers between users.

Status: Unavoidable (intrinsic feature of any system that allows transfers between users)

Update from the client: After discussion with the team, we concluded that only a few parties (initially us, and subsequently other builders benefiting from SWAY emissions) could gain from this. Due to how emissions adjust to target the long-term moving average, our modeling showed minimal potential net gain.

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 the SWAY documentation

The SWAY has provided documentation about the protocol through in-line comments. Additionally, the SWAY team was available to address any questions or concerns from the Nethermind Security team.

8 Test Suite Evaluation

The SWAY module is part of a system. The system has many test cases. In this section, we highlight the test cases related to SWAY .

```
#[test]
#[available_gas(1000000)]
fn test_constructor() {
    let caller = starknet::contract_address_const::<'ADMIN'>();
    starknet::testing::set_caller_address(caller);

    let mut state = Sway::contract_state_for_testing();
    Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
    let res = Sway::has_grant(@state, caller, Sway::roles::ADMIN);

    assert(res, 'deployer should be admin');
}
```

```
#[test]
#[available_gas(1000000)]
fn test_grants() {
    let caller = starknet::contract_address_const::<'ADMIN'>();
    starknet::testing::set_caller_address(caller);

    let mut state = Sway::contract_state_for_testing();
    Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);

    Sway::add_grant(ref state, caller, Sway::roles::ADMIN);
    let res = Sway::has_grant(@state, caller, Sway::roles::ADMIN);
    assert(res, 'caller should be minter');

    Sway::revoke_grant(ref state, caller, Sway::roles::ADMIN);
    let res = Sway::has_grant(@state, caller, Sway::roles::ADMIN);
    assert(!res, 'caller should not be minter');
}
```

```
#[test]
#[available_gas(1000000)]
#[should_panic(expected: ('ERC20: must be admin', ))]
fn test_mint_without_grant() {
    let caller = starknet::contract_address_const::<'ADMIN'>();
    starknet::testing::set_caller_address(caller);

    let mut state = Sway::contract_state_for_testing();
    Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);

    let other = starknet::contract_address_const::<'OTHER'>();
    starknet::testing::set_caller_address(other);

    Sway::mint(ref state, other, (42 * 1000000).into());
}
```

```
#[test]
#[available_gas(1000000)]
fn test_mint() {
    let caller = starknet::contract_address_const::<'ADMIN'>();
    starknet::testing::set_caller_address(caller);

    let mut state = Sway::contract_state_for_testing();
    Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
    Sway::add_grant(ref state, caller, Sway::roles::ADMIN);

    let amount: u256 = (42 * 1000000).into();
    Sway::mint(ref state, caller, amount);
    let res = Sway::balance_of(@state, caller);
    assert(res == amount, 'caller should have funds');
}
```

```
    #[test]
    #[available_gas(2000000)]
    fn test_transfer() {
        let caller = starknet::contract_address_const::<'ADMIN'>();
        starknet::testing::set_caller_address(caller);

        let mut state = Sway::contract_state_for_testing();
        Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
        Sway::add_grant(ref state, caller, Sway::roles::ADMIN);

        let amount: u256 = (42 * 1000000).into();
        Sway::mint(ref state, caller, amount);

        let receiver = starknet::contract_address_const::<'RECEIVER'>();
        Sway::transfer(ref state, receiver, amount);
        let res = Sway::balance_of(@state, receiver);
        assert(res == amount, 'receiver should be funded');
    }
```

```
    #[test]
    #[available_gas(2000000)]
    #[should_panic(expected: ('ERC20: insufficient allowance', ))]
    fn test_transfer_wrong_caller() {
        let caller = starknet::contract_address_const::<'ADMIN'>();
        starknet::testing::set_caller_address(caller);

        let mut state = Sway::contract_state_for_testing();
        Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
        Sway::add_grant(ref state, caller, Sway::roles::ADMIN);

        let amount: u256 = (42 * 1000000).into();
        Sway::mint(ref state, caller, amount);

        let receiver = starknet::contract_address_const::<'RECEIVER'>();
        let wrong_sender = starknet::contract_address_const::<'WRONG_SENDER'>();
        starknet::testing::set_caller_address(wrong_sender);
        Sway::transfer_from(ref state, caller, receiver, amount);
    }
```

```
    #[test]
    #[available_gas(2000000)]
    #[should_panic(expected: ('ERC20: insufficient balance', ))]
    fn test_insufficient_funds() {
        let caller = starknet::contract_address_const::<'ADMIN'>();
        starknet::testing::set_caller_address(caller);

        let mut state = Sway::contract_state_for_testing();
        Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
        Sway::add_grant(ref state, caller, Sway::roles::ADMIN);

        let amount: u256 = (42 * 1000000).into();
        Sway::mint(ref state, caller, amount);

        let receiver = starknet::contract_address_const::<'RECEIVER'>();
        Sway::transfer(ref state, receiver, (50 * 1000000).into());
    }
```

```
    #[test]
    #[available_gas(2000000)]
    fn test_send_with_approval() {
        let caller = starknet::contract_address_const::<'ADMIN'>();
        starknet::testing::set_caller_address(caller);
        let other_sender = starknet::contract_address_const::<'OTHER_SENDER'>();

        let mut state = Sway::contract_state_for_testing();
        Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
        Sway::add_grant(ref state, caller, Sway::roles::ADMIN);

        let amount: u256 = (42 * 1000000).into();
        Sway::mint(ref state, caller, amount);
        Sway::approve(ref state, other_sender, amount);

        starknet::testing::set_caller_address(other_sender);
        let receiver = starknet::contract_address_const::<'RECEIVER'>();
        Sway::transfer_from(ref state, caller, receiver, amount);

        let res = Sway::balance_of(@state, receiver);
        assert(res == amount, 'receiver should be funded');
    }
```

```
    #[test]
    #[available_gas(2000000)]
    fn test_transfer_with_confirmation() {
        let caller = starknet::contract_address_const::<'ADMIN'>();
        starknet::testing::set_caller_address(caller);

        let mut state = Sway::contract_state_for_testing();
        Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
        Sway::add_grant(ref state, caller, Sway::roles::ADMIN);

        let amount: u256 = (42 * 1000000).into();
        Sway::mint(ref state, caller, amount);

        let receiver = starknet::contract_address_const::<'RECEIVER'>();
        let consumer = starknet::contract_address_const::<'CONSUMER'>();
        Sway::transfer_with_confirmation(ref state, receiver, 42 * 1000000, 'memo1', consumer);

        starknet::testing::set_caller_address(consumer);
        Sway::confirm_receipt(ref state, caller, receiver, 42 * 1000000, 'memo1');

        let res = Sway::balance_of(@state, receiver);
        assert(res == amount, 'receiver should be funded');
    }
```

```
    #[test]
    #[available_gas(2000000)]
    fn test_bridge_from_l1() {
        let caller = starknet::contract_address_const::<'ADMIN'>();
        starknet::testing::set_caller_address(caller);

        let mut state = Sway::contract_state_for_testing();
        Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
        Sway::add_grant(ref state, caller, Sway::roles::ADMIN);

        Sway::set_l1_bridge(ref state, 'L1_BRIDGE'.try_into().unwrap());
        Sway::handle_deposit(
            ref state, 'L1_BRIDGE', starknet::contract_address_const::<'PLAYER'>(), 42000, 0, 0x123
        );

        let balance = Sway::balance_of(@state, starknet::contract_address_const::<'PLAYER'>());
        assert(balance == 42000, 'wrong balance');
    }
```

```
    #[test]
    #[available_gas(2000000)]
    #[should_panic(expected: ('Bridge: from l1 bridge only', ))]
    fn test_bridge_from_l1_fail() {
        let caller = starknet::contract_address_const::<'ADMIN'>();
        starknet::testing::set_caller_address(caller);

        let mut state = Sway::contract_state_for_testing();
        Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
        Sway::add_grant(ref state, caller, Sway::roles::ADMIN);

        Sway::set_l1_bridge(ref state, 'L1_BRIDGE'.try_into().unwrap());
        Sway::handle_deposit(
            ref state, 'WRONG_L1_BRIDGE', starknet::contract_address_const::<'PLAYER'>(), 42000, 0, 0x123
        );

        let balance = Sway::balance_of(@state, starknet::contract_address_const::<'PLAYER'>());
        assert(balance == 42000, 'wrong balance');
    }
```

```
    #[test]
    #[available_gas(2000000)]
    fn test_bridge_withdraw() {
        let caller = starknet::contract_address_const::<''>();
        starknet::testing::set_caller_address(caller);

        let mut state = Sway::contract_state_for_testing();
        Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
        Sway::add_grant(ref state, caller, Sway::roles::ADMIN);
        Sway::add_grant(ref state, caller, Sway::roles::ADMIN);
        Sway::mint(ref state, starknet::contract_address_const::<'PLAYER'>(), 42000.into());

        Sway::set_l1_bridge(ref state, 'L1_BRIDGE'.try_into().unwrap());

        starknet::testing::set_caller_address(starknet::contract_address_const::<'PLAYER'>());
        Sway::initiate_withdrawal(ref state, 'L1_ACCOUNT'.try_into().unwrap(), 21000.into());

        let balance = Sway::balance_of(@state, starknet::contract_address_const::<'PLAYER'>());
        assert(balance == 21000, 'wrong balance');
    }
```

8.1 Complementary Tests

We developed several test cases to validate the correct behavior of SWAY . To test the functionality of recording the volume of transfers per period, we created a mock `block.timestamp` variable. This allowed us to alter the current timestamp to ensure that the storage variable `period_volumes` is being properly populated. Below, we show the declaration of `period_volumes` and the new mock variable `_block_timestamp`.

```
#[storage]
struct Storage {
    ...
    period_volumes: LegacyMap::<u64, u256>,
    role_grants: LegacyMap::<(ContractAddress, u64), bool>,
    _block_timestamp: u64,
}
```

To set and read the `block.timestamp`, we created the functions below.

```
#[external(v0)]
fn current_period(ref self: ContractState) -> u64 {
    return self._block_timestamp.read() / RECORDING_PERIOD;
}

#[external(v0)]
fn set_current_period(ref self: ContractState, period: u64) {
    self._block_timestamp.write(period);
}
```

The test case below checks the infinite allowance and tries to transfer tokens between users without allowance. `SWAY` passes this test.

```
#[test]
#[available_gas(20000000)]
#[should_panic(expected: ('ERC20: insufficient allowance', ))]
fn test_unlimited_approval_new() {
    let caller = starknet::contract_address_const::<'ADMIN'>();
    starknet::testing::set_caller_address(caller);
    let user1 = starknet::contract_address_const::<'OTHER_SENDER'>();
    let user2 = starknet::contract_address_const::<'RECEIVER'>();

    let mut state = Sway::contract_state_for_testing();
    Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
    Sway::add_grant(ref state, caller, Sway::roles::ADMIN);

    // mint to caller
    let amount_1000: u256 = (1000).into();
    Sway::mint(ref state, caller, amount_1000);

    // approve caller -> user1, infinite

    let amount: u256 = 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff_u256;
    //let amount: u256 = 1000_u256;
    Sway::approve(ref state, user1, amount);

    // change caller to user1, who has allowance
    starknet::testing::set_caller_address(user1);
    Sway::transfer_from(ref state, caller, user2, amount_1000); // transfer from caller to user2
    let balance: u256 = Sway::balance_of(@state, user2); // check user2 balance
    assert(balance == amount_1000, 'balance error');

    let balance: u256 = Sway::balance_of(@state, user1); // check user1 balance
    assert(balance == (0).into(), 'balance error');

    let balance: u256 = Sway::balance_of(@state, caller); // check caller balance
    assert(balance == (0).into(), 'balance error');

    // check if allowance has changed
    let allow: u256 = Sway::allowance(@state, caller, user1);
    assert(allow == amount, 'receiver should be funded');
    starknet::testing::set_caller_address(user2);
    Sway::transfer(ref state,caller, amount_1000);

    // reset caller address
    starknet::testing::set_caller_address(caller);

    // change the approval to zero
    let amount: u256 = 0_u256;
    Sway::approve(ref state, user1, amount);

    // change caller to user1, who has zero allowance now. It should fail.
    starknet::testing::set_caller_address(user1);
    Sway::transfer_from(ref state, caller, user2, amount_1000); // transfer from caller to user2
}
```

The test case below checks the functionality of storing the amount transferred per period. `SWAY` also passes this test.

```
#[test]
#[available_gas(20000000)]
fn test_period_volumes_new() {
    let caller = starknet::contract_address_const::<'ADMIN'>();
    starknet::testing::set_caller_address(caller);

    let mut state = Sway::contract_state_for_testing();
    Sway::constructor(ref state, 'Standard Weighted Adalian Yield', 'SWAY', 6, caller);
    Sway::add_grant(ref state, caller, Sway::roles::ADMIN);

    // mint 1000 in time 0
    Sway::set_current_period(ref state, 0);
    let period: u64 = Sway::current_period(ref state);
    assert(period==0, 'error 1 period');
    let amount: u256 = (1000).into();

    Sway::mint(ref state, caller, amount);
    let res = Sway::balance_of(@state, caller);
    assert(res == 1000, 'not minted 1000 tokens');

    ////////////////////////////////////////////////
    /// performs two transfers in the same period
    ////////////////////////////////////////////////
    let amount = (50).into();
    let receiver = starknet::contract_address_const::<'RECEIVER'>();
    Sway::transfer(ref state, receiver, amount);
    let res = Sway::balance_of(@state, receiver);
    assert(res == 50, 'error a');

    Sway::set_current_period(ref state, 900_000);

    Sway::transfer(ref state, receiver, amount);
    let res = Sway::balance_of(@state, receiver);
    assert(res == 100, 'error b');

    // check value of period_volume
    let volume: u256 = Sway::get_period_volume(ref state, 0);
    assert(volume == 100, 'error 2');

    ////////////////////////////////////////////////
    /// change the period
    ////////////////////////////////////////////////
    Sway::set_current_period(ref state, 2_000_000);
    let volume: u256 = Sway::get_period_volume(ref state, 1);
    assert(volume == 0, 'error 3');

    let volume: u256 = Sway::get_period_volume(ref state, 2);
    assert(volume == 0, 'error 31');

    // transfer 50 tokens to receiver in time 2_000_000
    let amount = (50).into();
    let receiver = starknet::contract_address_const::<'RECEIVER'>();
    Sway::transfer(ref state, receiver, amount);
    let res = Sway::balance_of(@state, receiver);
    assert(res == 150, 'receiver should be funded');
    let volume: u256 = Sway::get_period_volume(ref state,2);
    assert(volume == 50, 'error 4');

    // caller balance should be 850
    let res = Sway::balance_of(@state, caller);
    assert(res == 850, 'error 41');
}

```

Remarks about `SWAY` tests

In summary, the protocol has a very complete test suite.

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 procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, 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

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