Security Review Report
NM-0250 Dinero Migration

NETHERMIND SECURITY

(Jul 19, 2024)

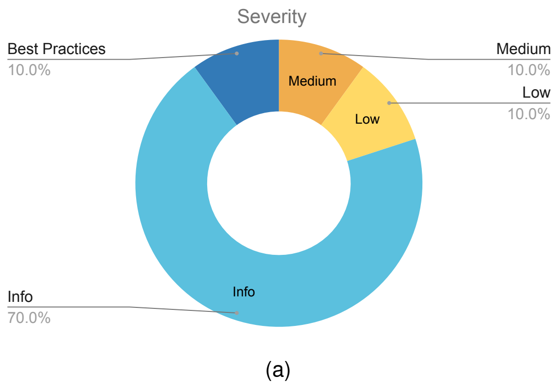
Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the Dinero. This security engagement focused on the migration process from the old governance token BTRFLYV2 to the new DINERO token. The new system introduces a StakedDinero vault, where the governance token holders can stake their tokens and earn DINERO rewards. The users can use the Migrator contract to burn their legacy tokens in exchange for the DINERO token. The pxBTRFLY and rpxBTRFLY holders that have unclaimed snapshot and futures rewards can still claim them using the Migrator contract even after the PirexBtrfly contract is winded down.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** seven points of attention, where one is classified as Medium, one is classified as Low, and five 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. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



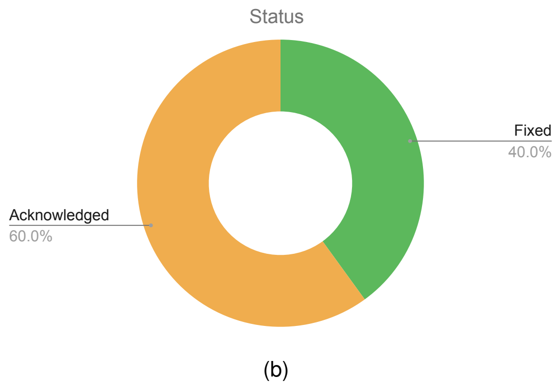


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (1), **Low** (1), **Undetermined** (0), **Informational** (7), **Best Practices** (1).
Distribution of status: Fixed (4), **Acknowledged** (6), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jun 28, 2024
Response from Client	Regular responses during audit engagement
Final Report	Jul 19, 2024
Repository	dinero
Commit (Audit)	fc14e4ba71b51d84346634214d7d86c623e69716
Commit (Final)	2577817f74fd7fc090f12958eed6cf7c83901650
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	Migrator.sol	303	183	60.4%	91	577
2	StakedDinero.sol	346	314	90.8%	116	776
3	DineroERC20Upgradeable.sol	36	34	94.4%	10	80
4	Dinero.sol	15	14	93.3%	2	31
5	libraries/Errors.sol	15	42	280.0%	12	69
6	libraries/DataTypes.sol	11	20	181.8%	3	34
	Total	**726**	**607**	**83.6%**	**234**	**1567**

3 Summary of Issues

	Finding	Severity	Update
1	Block any user action by depositing to their address	Medium	Fixed
2	Privileged burn of a `Dinero` token	Low	Acknowledged
3	Incorrect assumption during the redemption lock period	Info	Acknowledged
4	Migration contract does not terminate its function	Info	Acknowledged
5	Redemption should revert on zero supply	Info	Fixed
6	The `address(0)` on transfers	Info	Acknowledged
7	The function `permit(...)` does not protect from signature malleability	Info	Acknowledged
8	The implementation contracts can be initialized	Info	Fixed
9	Unclear error messages	Info	Acknowledged
10	Unused code	Best Practices	Fixed

4 System Overview

The **Dinero protocol** is planning a migration to a new governance token, `Dinero`, aimed at consolidating multiple tokens into a single unified token. The migration is facilitated through the `Migrator` contract, which manages the conversion process from the old governance token `btrflyV2` and its derivatives to the new `Dinero` token. The protocol introduces a `StakedDinero` vault where users can deposit their new `Dinero` tokens and receive `sDINERO` shares in return. Staking rewards are paid in `Dinero` and are deposited into the vault by the protocol team. The rewards are streamed over time based on the `rewardRate` updated by the privileged `Distributor` account.

The `Migrator` contract maintains the functionality to allow the existing `pxBtrfly` and `rpxBtrfly` holders to claim their unclaimed legacy snapshot and futures rewards.

The following sections delve into the system's components and their interactions. The diagram below showcases a high-level view of the system's architecture.

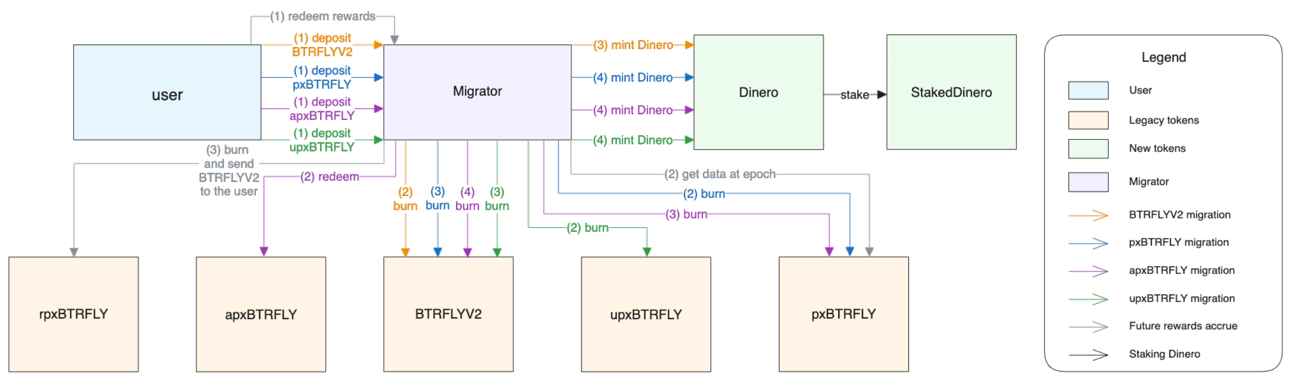


Fig. 2: Dinero migration overview.

4.1 Migration process

The migration process starts with a pause and an emergency withdrawal of the `BTRFLYV2` tokens from the `PirexBtrfly` contract. The tokens are transferred to the `Migrator` contract, which will handle the migration process. The `migrate(...)` function of the `Migrator` contract is the system's entry point for the users. Users must provide the amounts of each token they wish to migrate to `Dinero`. The tokes available for migration are: `BTRFLYV2`, `pxBTRFLY`, `apxBTRFLY` and `upxBTRFLY`. Derivative tokens are first brought to their basic form e.g. to `pxBTRFLY` in case of the `apxBTRFLY` and `BTRFLYV2` for `upxBTRFLY` and then burned alongside the `BTRFLYV2` tokens received from the `PirexBtrfly` contract. For each migrated `BTRFLYV2` token, the user receives `DINERO` tokens at a predefined ratio of `2_000` `DINERO` per `1` `BTRFLYV2` token.

When calling `migrate(...)`, the users have the option to decide whether they want to receive the `DINERO` tokens to their address or stake them in the `StakedDinero` vault. In the latter case, the users would receive the `sDINERO` vault share tokens instead. The next section explains the staking process in greater detail.

Once the `PirexBtrfly` contract is deactivated, the rights to process the reward claims are transferred to the `Migrator` contract. This allows the existing `pxBtrfly` and `rpxBtrfly` holders to claim their unclaimed legacy rewards even after the migration process is finished.

4.2 Actors of the system

The following actors are present within the system:

- **Owner**: The ownership of the `Dinero`, `StakedDinero` and `Migrator` contracts is held by the `governance` address. The owner is a privileged actor who can grant and revoke Minter and Burner roles, change key protocol parameters such as deposit and redemption lock duration, and change the rewards `Distributor` address.
- **Rewards Distributor**: The new rewards added to the `StakedDinero` contract need to be streamed over time. The distributor is responsible for updating the reward stream parameters, such as the reward rate and the reward period finish timestamp. This logic is handled inside the `notifyRewardAmount(...)` function from the `StakedDinero` contract.
- **Minter**: The holder of the Minter role can mint an arbitrary amount of `DINERO` tokens to any address. During the deployment, this role is granted exclusively to the `Migrator` contract.
- **Burner**: The burner can burn an arbitrary amount `DINERO` tokens from any address. This role is currently not utilized in the system. It is reserved for future expansions that may require the burning of `DINERO` tokens.
- **Users**: Based on the type of token that they hold, the users can be divided into the following groups:
  - **Legacy token holders**: Users holding the `BTRFLYV2`, `pxBTRFLY`, `apxBTRFLY` or `upxBTRFLY` can use the `Migrator` contract to exchange these tokens for the new `DINERO` token.

– **Users with unclaimed rewards**: Users that have unclaimed snapshot rewards (for holding `pxBTRFLY`) or futures rewards (`rpxBTRFLY` holder) can use the `Migrator` contract to claim their reward tokens once the old `PirexBtrfly` system is winded down.

– **Dinero token holders**: Users holding the new `DINERO` token can use the `StakedDinero` vault to stake their tokens and receive the `sDINERO` vault share tokens in return. Over time, as rewards are streamed, the assets accumulate inside the vault, and users can redeem their shares for more `DINERO` tokens. The staking process is explained in greater detail in the following section.

4.3 Staking Dinero tokens

Once the `DINERO` token is obtained by the user, it can be deposited into the `StakedDinero` vault to earn the rewards. The entry point for the user is the `deposit(...)` function. For the provided `DINERO` assets, the user receives the `sDINERO` vault shares, which can later be redeemed back for `DINERO` tokens.

The redemption is a two-step process. First, the users call the `initiateRedemption(...)` function, which burns their `sDINERO` shares and puts their redemption into a queue. After a certain amount of time, users can call the `redeem(...)` function to finalize the redemption process. If a user has multiple redemptions waiting, he can redeem them all at once. The `redeem(...)` function will iterate over the records from the queue and send the `DINERO` tokens all at once.

The `StakedDinero` vault implements a soft-lock mechanism that prevents users from initiating redemptions immediately after making a deposit or from finalizing the redemptions right after initiating them. Currently, both redemption and deposit lock durations are set to `1` day during the contract's initialization. After the initial deployment, they can be set to any value different than `0` by the contract owner.

The staking rewards are paid in `DINERO` tokens and are streamed over the period of `7` days. Once the new rewards are sent to the `StakedDinero` contract, a special address called Distributor will call the `notifyRewardAmount(...)` function to update the rewards rate for the next period.

5 Risk Rating Methodology

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6 Issues

6.1 [Medium] Block any user action by depositing to their address

File(s): StakedDinero.sol, Migrator.sol

Description: After depositing Dinero tokens to the StakedDinero contract, the recipient's actions are blocked for the period defined in the depositLockDuration, which initially is set to 1 day. This is achieved by adding the checkDepositLock(...) modifier to functions: transfer(...), transferFrom(...) and initiateRedemption(...). The malicious actor can prevent any user from transferring or redeeming tokens by specifying the victim address in the receiver parameter during the call to the deposit(...) function:

```
function deposit(
    uint256 assets,
    // @audit: any address can be provided
    address receiver
) external nonReentrant returns (uint256) {
    address sender = msg.sender;
    uint256 shares = previewDeposit(assets);
    // ...
    StakedDineroStorage storage $ = _getStakedDineroStorage();
    // @audit: the lock is refreshed for the specified address
    $.depositLockExpiryTime[receiver] =
        block.timestamp +
        $.depositLockDuration;
    // ...
    $.asset.safeTransferFrom(sender, address(this), assets);

    _mint(receiver, shares);
    _stake(assets);

    return shares;
}
```

As a result, an attacker who deposits assets with the victim's address as a receiver may block any user from transferring/redeeming their funds. Note that this would require the attacker to lose their own assets, and there is no monetary gain for the attacker.

Recommendation(s): Since the StakedDinero should receive the direct deposit as well as deposits through the Migrator, consider applying the following changes to fix the issue:

– in the StakedDinero.deposit(...), if the msg.sender is not the Migrator, use msg.sender to mint and update the lock;

– if the msg.sender is the Migrator use the receiver parameter to mint and update the lock;

– in the Migrator.migrate(...), use msg.sender instead of receiver parameter to mint and deposit funds to StakedDinero;

Status: Fixed

Update from the Nethermind Security: Current solution fixes the issue. Consider clearly documenting that the stake of migrated tokens is done for msg.sender, not the _receiver.

6.2 [Low] Privileged burn of a Dinero token

File(s): DineroERC20Upgradeable.sol

Description: The Dinero token is a new version of a BtrflyV2 governance token. However, the Dinero token can be burned by the BURNER_ROLE from any account. This is not only a difference compared to the previous token but introduces unnecessary risk for all the Dinero holders.

Recommendation(s): Consider removing unnecessary privileged burns to decrease the attack surface and allow users to burn their Dinero tokens on their own.

Status: Acknowledged

Update from the client: We will keep the role as this may be used for future expansion of futures requiring the burning of Dinero.

6.3 [Info] Incorrect assumption during the redemption lock period

File(s): `StakedDinero.sol`

Description: The `initiateRedemption(...)` function is called by the users to start the redemption process. For the provided `sDINERO` token shares, the function computes the amount of `DINERO` tokens that the user will get in return and burns the provided shares. The redemption is added to the `PendingRedemptionQueue` data structure, which holds the redemption amounts and timestamps for when the redemption can be finalized.

```
function initiateRedemption(uint256 shares)
    external
    checkDepositLock(msg.sender)
    returns (uint256)
{
    uint256 assets = previewInitiateRedemption(shares);
    // ...
    _burn(msg.sender, shares);
    // ...
    // @audit User can redeem the assets once the activeTime is reached.
    redemption.activeTime = block.timestamp + $.redemptionLockDuration;
    redemption.amount = assets;
    queue.pendingRedemptions.push(redemption);
    // ...
}
```

The `redeem(...)` function is the last step of the redemption process. It iterates over the queue elements to check which pending redemptions are ready to be redeemed. It sums up the individual redemption amounts and stops at the first redemption that cannot be redeemed yet. After that, the calculated amount of `DINERO` is transferred to the user.

```
function redeem(...)
    external
    nonReentrant
    returns (uint256)
{
    // ...
    uint256 currentTime = block.timestamp;

    for (uint256 i = count; i < queueLen; ++i) {
        DataTypes.PendingRedemption memory record = redemptions[i];
        // @audit If the redemption cannot be processed yet,
        // go to the `else` branch and break.
        if (record.activeTime <= currentTime && limit > 0) {
            redeemAmount += record.amount;
            ++count;
            --limit;
        } else {
            // Can safely break here since subsequent records will have
            // further expiry time
            // or if we hit the max number of processed records
            break;
        }
    }
    // ...
    $.asset.safeTransfer(receiver, redeemAmount);
    // ...
}
```

One of the assumptions made in the `redeem(...)` function is that the subsequent redemption records will have the `activeTime` set further into the future. However, this assumption might be broken in a situation where the owner decreases the `redemptionLockDuration` by calling the `setRedemptionLockDuration(...)` function. Consider the following scenario:

1. User initiates 1st redemption while the `redemptionLockDuration` is set to `24 hours`;
2. The owner decreases the `redemptionLockDuration` to `1 hour`;
3. The user initiates 2nd redemption (lock time is `1 hour`);
4. After `1 hour`, the user attempts to `redeem(...)` the 2nd redemption;

Since the 1st redemption cannot be redeemed yet, the `for` loop will stop at the 1st record, and the 2nd redemption won't be processed even though its redemption time was already reached.

Recommendation(s): The described issue may be resolved by not breaking the loop at the first record that does not meet the time lock condition but iterating over all records up to the limit or end of the array.

Status: Acknowledged.

Update from the Nethermind Security: The changes introduced in 635a9f1 create a new problem. Consider that the `redeem(...)` function is called, and the `redemptions` array contains elements: `[a, b]`, where `a` is locked, but `b` is unlocked. The current implementation with the applied changes allows for using `b`, even if the previous element `a` is locked. In that case, the `count` would be increased, and the `a` element would be unreachable. Additionally, during the next call, the `b` would be used again. The current design that involves the `count` variable at the beginning of the iteration makes the correct solution for the described problem complex and costly. Therefore consider:

- instead of using `count` as the beginning of the iteration, use user-defined indexes to loop over chosen elements in the array and mark the used elements, similarly to `Migrator._redeemSnapshotRewards(...)`;
- alternatively, use the original version since the potential issue described is low severity and would happen very rarely;

Update from the client: Changed to original version at ce2eec2.

6.4 [Info] Migration contract does not terminate its function

File(s): `Migrator.sol`

Description: The `Migrator` allows for exchanging the inactive tokens for the `Dinero` token. However, there is no possibility of ceasing the functioning of the `Migrator` contract. Lack of inactivation of the `Migrator` increases the potential attack surface. One potential problem is that the private keys of the owners of the `Migrator`, inactive tokens, and `PirexBtrfly` have to be securely maintained by the team, even if those contracts are no longer used, since `Migrator` is a minter of a `Dinero` token.

Recommendation(s): Consider bounding the time in which users can use the `Migrator` to mint new `Dinero` tokens to decrease the attack surface.

Status: Acknowledged.

6.5 [Info] Redemption should revert on zero supply

File(s): `StakedDinero.sol`

Description: The `previewInitiateRedemption(...)` function is used to compute the amount of `DINERO` tokens that the user will receive for provided `sDINERO` shares. The return value depends on the total supply of the `sDINERO` tokens. If the total supply is `0`, then the output amount is equal to the provided input. This is incorrect since if there are no shares in circulation, then it should not be possible to redeem/initiate redemption in the first place.

```
function initiateRedemption(
    uint256 shares
) external checkDepositLock(msg.sender) returns (uint256) {
    // @audit: return assets==shares if the totalSupply=0
    uint256 assets = previewInitiateRedemption(shares);

    if (assets == 0) revert Errors.ZeroAmount();

    // ...
    // @audit: revert on burn, since there are no shares to burn
    _burn(msg.sender, shares);

    // ...
}
```

As a result, the function would revert on burn, which is not a clear error message for the caller.

Recommendation(s): If the `totalSupply=0`, consider either reverting in the `previewInitiateRedemption(...)` with a custom error message or return zero, which will revert with `Errors.ZeroAmount()` in the `initiateRedemption(...)` function.

Status: Fixed.

6.6 [Info] The `address(0)` on transfers

File(s): `DineroERC20.sol`

Description: The functions `transfer(...)` and `transferFrom(...)` allow for passing `to` and `from` parameters with value `address(0)`. However, `address(0)` is a special address and should be only used in `mint(...)` and `burn(...)`. One direct issue of allowing `address(0)` in transfers is that transfer to `address(0)` and burn would emit the same event `Transfer(sender, address(0), amount)`.

Recommendation(s): Consider checking if values of `to` and `from` are not `0` in the `transfer(...)` and `transferFrom(...)` functions.

Status: Acknowledged

6.7 [Info] The function `permit(...)` does not protect from signature malleability

File(s): `DineroERC20.sol`

Description: The function `permit(...)` does not restrict the range of the `s` value, allowing for the creation of a new valid signature given the original one. This is currently not a problem because the signatures are not stored or used for any additional purpose.

Recommendation(s): Consider clearly documenting the fact that signatures used in `permit(...)` should not be used for any other purpose because of their malleability.

Status: Acknowledged

6.8 [Info] The implementation contracts can be initialized

File(s): Dinero.sol, StakedDinero.sol

Description: The implementation contracts `Dinero` and `StakedDinero` do not disable the `initialize(...)` functions. This allows anyone to call `initialize(...)` with arbitrary values, which may lead to unwanted behavior. One example of such malicious behavior is initiating a call from the implementation contracts to sanctioned or black-listed contracts. As a result, the anti-money laundering (AML) systems may flag the implementation contracts as suspicious.

Recommendation(s): Consider removing the possibility of calling the initializers directly on the implementation contracts. This may be done by calling the `_disableInitializers(...)` in the constructor, which will disable calling functions with the `initializer` modifier. This is already done in the `Migrator` contract, so a similar approach can be taken for the `Dinero` and `StakedDinero` contracts.

Status: Fixed.

6.9 [Info] Unclear error messages

File(s): `DineroERC20.sol`

Description: Important actions like transfers are not handled with clear error messages. As a result, on a try of a transfer of a higher amount than in balance, the error message would be "uint256 overflow." The lack of clear error messages may make it difficult to monitor on-chain activity and provide clear messages to the user.

Recommendation(s): Consider implementing clear error messages.

Status: Acknowledged

6.10 [Best Practices] Unused code

File(s): `Errors.sol`

Description: The unused can affect overall readability and code quality. The `ExceedsMax()` error from the `Errors` library is currently unused.

Recommendation(s): Consider removing the unused error.

Status: Fixed.

7 Documentation Evaluation

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

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

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

Remarks about the Dinero documentation

The documentation for the Dinero Migration is contained in the inline code comments.
The team answered every question during meetings or through messages, which gave the auditing team a lot of insight and a deep understanding of the technical aspects of the project

8 Test Suite Evaluation

```
./src/scripts/forgeTest.sh
[] Compiling...
[] Compiling 107 files with Solc 0.8.25
[] Solc 0.8.25 finished in 6.44s
Compiler run successful!

Ran 8 tests for test/Dinero.t.sol:DineroTest
[PASS] testBurn(uint224) (runs: 100, : 56129, ~: 56142)
[PASS] testCannotBurnNoBurnerRole() (gas: 16950)
[PASS] testCannotBurnZeroAddress() (gas: 13889)
[PASS] testCannotBurnZeroAmount() (gas: 13833)
[PASS] testCannotMintNoMinterRole() (gas: 16961)
[PASS] testCannotMintZeroAddress() (gas: 13768)
[PASS] testCannotMintZeroAmount() (gas: 13866)
[PASS] testMint(uint224) (runs: 100, : 66173, ~: 66173)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 3.65s (37.91ms CPU time)

Ran 26 tests for test/StakedDinero.t.sol:StakedDineroTest
[PASS] testCannotDepositZeroAddress() (gas: 34943)
[PASS] testCannotDepositZeroAmount() (gas: 34897)
[PASS] testCannotInitiateRedemptionLocked() (gas: 146652)
[PASS] testCannotInitiateRedemptionZeroAmount() (gas: 34563)
[PASS] testCannotNotifyRewardAmountUnathorised() (gas: 34919)
[PASS] testCannotNotifyRewardAmountnoRewards() (gas: 51049)
[PASS] testCannotRedeemNotActive(uint96) (runs: 100, : 231940, ~: 231940)
[PASS] testCannotRedeemZeroAddress() (gas: 16531)
[PASS] testCannotRedeemZeroAmount() (gas: 20895)
[PASS] testCannotSetDepositLockDurationUnauthorised() (gas: 34201)
[PASS] testCannotSetDepositLockDurationZeroAmount() (gas: 13587)
[PASS] testCannotSetDistributorUnauthorised() (gas: 34277)
[PASS] testCannotSetDistributorZeroAddress() (gas: 13732)
[PASS] testCannotSetRedemptionLockDurationUnauthorised() (gas: 34148)
[PASS] testCannotSetRedemptionLockDurationZeroAmount() (gas: 13565)
[PASS] testCannotTransferLocked(bool) (runs: 100, : 147412, ~: 147408)
[PASS] testDeposit(uint96) (runs: 100, : 155550, ~: 155550)
[PASS] testDepositWithPendingRewards(uint96,uint96,uint32) (runs: 100, : 310505, ~: 310473)
[PASS] testHarvest(uint96,uint32) (runs: 100, : 197709, ~: 197705)
[PASS] testInitiateRedemption(uint96) (runs: 100, : 236520, ~: 236520)
[PASS] testNotifyRewardAmountWithPendingRedemptions(uint96,uint96) (runs: 100, : 309764, ~: 309764)
[PASS] testRedeem(uint96) (runs: 100, : 278059, ~: 278058)
[PASS] testSetDepositLockDuration() (gas: 22434)
[PASS] testSetDistributor() (gas: 20008)
[PASS] testSetRedemptionLockDuration() (gas: 22342)
[PASS] testTransfer(bool) (runs: 100, : 183168, ~: 183168)
Suite result: ok. 26 passed; 0 failed; 0 skipped; finished in 46.11s (273.53ms CPU time)

Ran 8 tests for test/Migrator.t.sol:MigratorTest
[PASS] testBulkRedeemFuturesRewards() (gas: 444511)
[PASS] testBulkRedeemSnapshotRewards() (gas: 476500)
[PASS] testCannotMigrateInsufficientBalance() (gas: 24032)
[PASS] testCannotMigrateZeroAmount() (gas: 18566)
[PASS] testMigrate(uint96) (runs: 100, : 144229, ~: 144229)
[PASS] testMigrateWithPxTokens(uint96,uint96) (runs: 100, : 663016, ~: 663016)
[PASS] testMigrateWithUpxTokens(uint96) (runs: 100, : 688238, ~: 688238)
[PASS] testRedeemSnapshotRewards() (gas: 444417)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 46.11s (91.96s CPU time)

Ran 3 test suites in 47.07s (95.86s CPU time): 42 tests passed, 0 failed, 0 skipped (42 total tests)
```

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