Security Review Report
NM-0930-Melllow



NETHERMIND
SECURITY

(May 25, 2026)

Contents

1 Executive Summary **2**

2 Audited Files **3**

3 Summary of Issues **3**

4 System Overview **4**
4.1 Initialization and Subvault Custody . . . 4
4.2 Request Allowlisting . . . 4
4.3 Push Flow . . . 4
4.4 Pull Flow . . . 4
4.5 Position Redemption . . . 5

5 Risk Rating Methodology **6**

6 Issues **7**
6.1 [Low] push() has no caller-supplied slippage bounds on the Request's authorization . . . 7
6.2 [Info] Unprotected `initialize(...)` function allows a Conditional Proxy Initialization Machine Parking (CPIMP) attack . . . 7
6.3 [Info] Yield-floor check in onRequestConsumed rounds toward zero, silently passing for small partial fills . . . 8
6.4 [Info] cancelOffers is gated by the allow set, blocking nonce cleanup after a disallow . . . 8
6.5 [Info] onRequestConsumed does not fully re-validate offer parameters . . . 9

7 Documentation Evaluation **10**

8 Test Suite Evaluation **11**
8.1 Tests Output . . . 11

9 About Nethermind **13**

1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security for Mellow's `ThreeFModule`, a Subvault module that brokers a Mellow Flexible-Vault's interaction with the external 3F Protocol.

The 3F Protocol is a dual-token funding marketplace in which each funding request issues a Principal Token (PT) and a Yield Token (YT). The module holds the Subvault's underlying asset and stands in as the marketplace counterparty for that Subvault. The asset-moving surface is gated along two axes:

- **Allow-set and whitelist gating**: Every fund-moving entry point routes through `_checkRequest`, which requires the target Request to sit in the module's local allow set and to currently report `Whitelisted` on the external `IWhitelist` registry. The allow set is mutated only by `ALLOW_REQUEST_ROLE`, after an asset match and a factory-identity check.

- **Push and pull flows under separated roles**: `PUSH_ROLE` consumes a Request's pre-granted `mintAuthorization` to mint PT and YT against the underlying asset. `PULL_ROLE` publishes a standing EIP-712 offer that the module attests to via EIP-1271; the 3F consumer pulls on demand, the Request settles through the `onRequestConsumed` callback, and a nonce advance closes the offer.

`BURN_ROLE` redeems a settled position by calling `burn(request)` once the Request reports `isRepaid` and `canWithdraw`, exchanging the module's PT and YT for the underlying asset via `burnAll`. `burn` is the one fund-moving entry point that skips the allow-set check, so a Request whose allow-set entry has been removed mid-lifecycle can still be unwound once it is repaid.

The audit comprises 248 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report five points of attention, where one is classified as `Low`, and four are classified as `Informational` severity. The issues are summarized in Fig. 1.

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

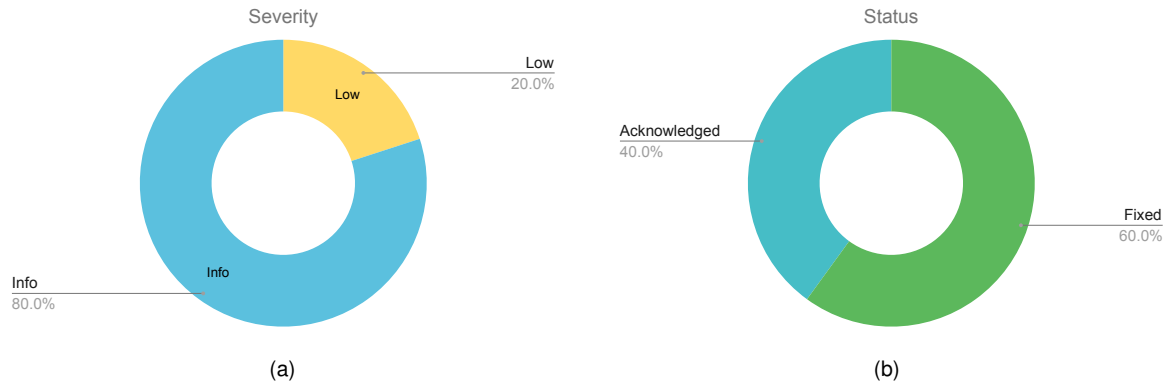


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (1), **Undetermined** (0), **Informational** (4), **Best Practices** (0). **Distribution of status: Fixed** (3), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	May 22, 2026
Final Report	May 25, 2026
Initial Commit	2029bf4
Final Commit	4bd08e4
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	ThreeFModule.sol	248	47	19%	44	339
	Total	**248**	**47**	**19%**	**44**	**339**

3 Summary of Issues

	Finding	Severity	Update
1	push() has no caller-supplied slippage bounds on the Request's authorization	Low	Fixed
2	Unprotected `initialize(...)` function allows a Conditional Proxy Initialization Machine Parking (CPIMP) attack	Info	Acknowledged
3	Yield-floor check in onRequestConsumed rounds toward zero, silently passing for small partial fills	Info	Acknowledged
4	cancelOffers is gated by the allow set, blocking nonce cleanup after a disallow	Info	Fixed
5	onRequestConsumed does not fully re-validate offer parameters	Info	Fixed

4 System Overview

The Mellow ThreeFModule is a Subvault-side custody module that connects a Mellow Flexible-Vault to the external 3F Protocol, a dual-token funding marketplace in which each Request issues a Principal Token (PT) against supplied capital and a Yield Token (YT) against future yield. The module holds the Subvault's underlying ERC-20 and brokers every fund-moving interaction between the Subvault and a 3F Request. All such entry points are gated by a role from the inherited `MellowACL` contract, and every Request the module touches must first be admitted to a local allow set anchored to the external 3F whitelist.

4.1 Initialization and Subvault Custody

The module is asset-specific and is initialized exactly once by a factory.

- **Constructor Binding**: The constructor takes `name_`, `version_`, and `asset_`. `asset` is stored as an immutable, and `SlotLibrary.getSlot(...)` derives an EIP-7201 storage slot from the deployment identity so each version label isolates its own state.
- **Initialization**: `initialize(bytes)` decodes `(admin, subvault, whitelist, requestFactory, holders, roles)`, writes the three external addresses into namespaced storage, grants `DEFAULT_ADMIN_ROLE` to `admin`, and zips `holders`/`roles` to assign `ALLOW_REQUEST_ROLE`, `PUSH_ROLE`, `PULL_ROLE`, and `BURN_ROLE`. `subvault`, `whitelist`, and `requestFactory` have no on-chain setters after this point.
- **Subvault Plane**: `pushAssets(value)` and `pullAssets(value)` are the only paths for `asset` to enter or leave the module to or from the Subvault. Both are restricted by the `onlySubvault` modifier, which checks `_msgSender() == subvault`.

4.2 Request Allowlisting

Before a 3F Request can be touched on any of the push, pull, or settlement paths, it must be admitted to a local allow set.

- **Admission**: `allowRequest(request)`, gated on `ALLOW_REQUEST_ROLE`, requires the Request to return `asset()` equal to the module's bound asset, to be recognized by `IRequestFactory.isRequest`, and to currently report `Whitelisted` on the external `IWhitelist` registry.
- **Per-Call Re-Check**: The fund-moving paths gated below (`push`, `authorizeOffer`, `onRequestConsumed`, `cancelOffer`) route through `_checkRequest`, which re-verifies allow-set membership and re-queries the whitelist on each call. `disallowRequest` or a whitelist revocation immediately closes those paths for the affected Request.

4.3 Push Flow

The push flow commits module-held capital to a Request that has pre-authorized this module as a minter.

- **Authorization Read**: `push(request)`, gated on `PUSH_ROLE`, reads `IRequest.mintAuthorization(this)` to retrieve the Request-side caps `(maxPt, minYt)`. The call reverts when both are zero or when the module's `asset` balance is below `maxPt`.
- **Mint and Delta Check**: The module `forceApprove`s `maxPt` of `asset` to the Request, calls `IRequest.mint(maxPt, minYt)`, and immediately revokes the approval. Pre- and post-`balancesOf` deltas are then compared against the caps; the call reverts on `ptMinted > maxPt` or `ytMinted < minYt`.

4.4 Pull Flow

The pull flow lets a 3F consumer initiate settlement at a time of its choosing against a standing offer attested by the module.

- **Offer Authorization**: `authorizeOffer(request, maxPt, minYt, duration)`, gated on `PULL_ROLE`, builds an `Offer` struct with `maker = address(this)`, an incremented nonce, and `expiration = block.timestamp + duration`. It hashes the offer under the Request's EIP-712 domain (read live from `IERC5267.eip712Domain()`) and writes `authorizedOffers[hash] = (maxPt, minYt)`.
- **ERC-1271 Attestation**: `isValidSignature(hash, ...)` returns the magic value for any hash present in `authorizedOffers`, allowing the Request to verify the offer through the EIP-1271 path on `consume(offer, sig, principal)`.
- **Settlement Callback**: When `consume` succeeds on the Request side, the Request calls back into `onRequestConsumed(offer, ..., principal, yield)`. The module re-derives the offer hash, requires the stored authorization to exist, enforces `principal <= maxPt` and the proportional yield floor `yield >= mulDiv(minYt, principal, maxPt)`, deletes the authorization, and `forceApprove`s `principal` of `asset` so the Request can pull payment on return.
- **Cancellation**: `cancelOffer(request)`, gated on `PULL_ROLE`, calls `IOfferReceiver.setNonce(nextNonce)` on the Request, invalidating every authorized offer issued at or below the previous nonce.

4.5 Position Redemption

A position is closed once the underlying funding is repaid.

- **Eligibility**: `burn(request)`, gated on `BURN_ROLE`, requires the Request to come from the registered `IRequestFactory`, to match the module's asset, and to report both `isRepaid()` and `canWithdraw()` true.
- **Settlement**: The module calls `IRequest.burnAll(this, this)` to redeem all held PT and YT into the underlying asset, then clears any residual ERC-20 allowance to the Request.
- **Allow-Set Bypass**: `burn` deliberately skips `_checkRequest`, so a Request whose allow-set entry has been removed mid-lifecycle can still be unwound once the repayment conditions are met.

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 [Low] push() has no caller-supplied slippage bounds on the Request's authorization

File(s): src/utils/ThreeFModule.sol

Description: `push(request)` reads `mintAuthorization(this)` from the Request at call time and forwards the returned `(maxPt, minYt)` straight into `mint()`, with no caller-supplied cap on `maxPt` or floor on `minYt`. If the Request's admin updates the authorization between an operator's observation and tx inclusion, the operator pays whatever the Request currently offers; the post-mint slippage check passes because it compares against the same values just read. Worst case, the module spends up to its full `asset` balance at a less favorable PT/YT ratio than the operator expected, bounded by what the allow-listed Request admin is willing to configure.

```
function push(address request) external nonReentrant onlyRole(PUSH_ROLE) {
    _checkRequest(request);
    address _this = address(this);
    // @audit-issue No caller-supplied cap on maxPt or floor on minYt; the on-chain authorization flows directly into
    ↪  mint().
    (uint128 maxPt, uint128 minYt) = IRequest(request).mintAuthorization(_this);
    // ....
    IERC20(asset).forceApprove(request, maxPt);
    IRequest(request).mint(maxPt, minYt);
    // ....
}
```

Recommendation(s): Consider adding caller-supplied `maxPtCap` and `minYtFloor` parameters to `push()` so the operator can pin worst-acceptable terms off-chain, with the function reverting when the on-chain authorization exceeds the cap or falls below the floor before any allowance is granted.

Status: Fixed

Update from the client: Updated to push(address request, uint128 maxPt, uint128 minYt). Two pre-mint guards added before forceApprove/mint(): if (authPt == 0 && authYt == 0) revert InsufficientAuthorization() rejects a missing authorization, and if (authPt > maxPt || authYt < minYt) revert InsufficientAuthorization() enforces the caller-supplied bounds. The existing post-mint defensive check if (ptMinted > authPt || ytMinted < authYt) revert SlippageExceeded() is retained.

6.2 [Info] Unprotected `initialize(...)` function allows a Conditional Proxy Initialization Machine Parking (CPIMP) attack

File(s): src/utils/ThreeFModule.sol

Description: The CPIMP attack exploits the window between proxy deployment and initialization. When a proxy is deployed with an empty `_data` field and `initialize()` is called in a separate transaction, an attacker monitoring the mempool can frontrun the initialization, insert a malicious intermediary implementation, and retain persistent backdoor access — surviving even future legitimate upgrades.

Read more about this attack at CPIMP Attack

Recommendation(s):

1. Maintain the atomic deployment pattern (best solution);
2. Verify the implementation slot post-deployment;

Status: Acknowledged

Update from the client: We are going to use src/factories/Factory.sol to have atomic deployment.

6.3 [Info] Yield-floor check in onRequestConsumed rounds toward zero, silently passing for small partial fills

File(s): src/utils/ThreeFModule.sol

Description: ThreeFModule.onRequestConsumed validates that the Request paid enough yield for the principal it pulled, by computing a proportional floor from the operator-set minYt: `yield >= mulDiv(minYt, principal, offer.amount)`. Because OZ Math.mulDiv rounds toward zero, the floor evaluates to 0 whenever `minYt * principal < offer.amount` i.e., whenever `principal < offer.amount / minYt`. In that window the check passes for any yield >= 0, including zero. The Consumer (the entity calling `request.consume(offer, sig, principal)`) chooses `principal` freely, so they can deliberately sit inside the rounding window. The Request then decides `yield`; if it honestly mirrors the same rounding the module ends up with the correct (zero) yield and nothing is stolen, but if a buggy or hostile Request mints sub-proportional yield this check fails to catch it. The module's floor check is the only on-module backstop for under-yielding by an allowed Request, so a rounding hole in it removes that backstop in the small-principal regime. Practical exploitation is bounded hard by gas economics as the rounding window is at most `offer.amount / minYt` wei per call, which for any meaningful yield rate is so small that draining a non-trivial fraction of an offer would cost orders of magnitude more in gas than the extracted value.

```
// ....
// @audit Math.mulDiv rounds down; floor = 0 when principal * minYt < offer.amount
if (yield < auth.minYt.mulDiv(principal, offer.amount)) {
    revert InsufficientYt();
}
```

Recommendation(s): Consider reformulating the yield-floor check so that integer division is eliminated entirely. The equivalent multiplied form, requiring the product of yield and offer.amount to be at least the product of minYt and principal, gives a bit-exact comparison with no precision loss.

Status: Acknowledged

Update from the client: The original formula from 3F protocol is used in the code.

6.4 [Info] cancelOffers is gated by the allow set, blocking nonce cleanup after a disallow

File(s): src/utils/ThreeFModule.sol

Description: `cancelOffers` calls `_checkRequest` as its first statement. After `disallowRequest(X)` removes X from `allowedRequests`, every `cancelOffers(X, _)` reverts with `RequestNotAllowed`. The operator's natural cleanup sequence (disallow, then cancel) is broken at the cancel step. To advance the Request's nonce afterward, the operator must either re-allow it, cancel, then disallow again, or coordinate off-chain with the Request's owner. No funds are at risk because `onRequestConsumed` also gates on `_checkRequest` and rejects `consume()` on a disallowed request. The cost is operational: outstanding offers stay attested via ERC-1271 (see F003) and outstanding nonces on the Request side stay unbumped.

```
// ....
function cancelOffers(address request, uint256 targetNonce) external nonReentrant onlyRole(PULL_ROLE) {
    // @audit-issue _checkRequest reverts when request is not in allowedRequests, so cancelOffers is unusable after
    ↪  disallowRequest.
    _checkRequest(request);
    // ....
}
```

Recommendation(s): Consider dropping the allow-set gate from `cancelOffers`, since the function's purpose is cleanup of requests no longer trusted. A lighter check, for example factory membership via `IRequestFactory.isRequest`, would still prevent PULL_ROLE from invoking `setNonce` on unrelated contracts. Alternatively, consider folding the nonce bump into `disallowRequest` itself so cleanup happens atomically.

Status: Fixed

Update from the client: Internal `_cancelOffers`(address request, uint256 targetNonce) is introduced. disallowRequest calls `_cancelOffers`-(request, lastIssuedNonce[request]), atomically advancing the on-chain nonce to the highest issued nonce and emitting OfferCancelled when offers are pending (no-op otherwise). Public cancelOffers retains `_checkRequest` since it is intended for cleanup of active, still-trusted requests; post-disallow nonce advancement is handled by disallowRequest itself.

6.5 [Info] onRequestConsumed does not fully re-validate offer parameters

File(s): src/utils/ThreeFModule.sol

Description: `onRequestConsumed` re-derives the offer hash and re-checks the request, the principal cap, the yield floor, and the asset balance, but two offer-parameter invariants are never asserted on the callback path:

1. `principal` is not required to be non-zero;
2. `offer.expiration` is not compared against `block.timestamp`;

```
// ....
function onRequestConsumed(Offer calldata offer, bytes calldata, uint256 principal, uint256 yield)
    external
    nonReentrant
{
    // @audit (1) principal == 0 bypasses all three guards below;
    // @audit (2) offer.expiration never checked against block.timestamp;
}
```

Recommendation(s): Add both missing guards at the top of the validation block, before the authorization lookup:

```
if (principal == 0) {
  revert ZeroValue();
}
if (block.timestamp > offer.expiration) {
  revert OfferExpired();
}
```

Status: Fixed

Update from the client: Both guards added at the top of the validation block, before the authorization lookup: if (principal == 0) revert ZeroValue() and if (block.timestamp > offer.expiration) revert OfferExpired().

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 Mellow's contracts documentation

The development team provided an overview of the system, its architecture and flows. The team was collaborative during the audit, providing answers to all questions raised by the Nethermind Security team through asynchronous communications.

8 Test Suite Evaluation

8.1 Tests Output

```
> forge test --match-path "test/unit/ThreeFModule.t.sol"
[] Compiling...
No files changed, compilation skipped

Ran 100 tests for test/unit/ThreeFModule.t.sol:ThreeFModuleTest
[PASS] testAllowRequest_AlreadyAllowed() (gas: 43559)
[PASS] testAllowRequest_AssetMismatch() (gas: 990833)
[PASS] testAllowRequest_Happy() (gas: 1055765)
[PASS] testAllowRequest_NotRole() (gas: 23201)
[PASS] testAllowRequest_RequestNotWhitelisted() (gas: 976117)
[PASS] testAllowedRequestAt() (gas: 20142)
[PASS] testAllowedRequestsCount() (gas: 1053270)
[PASS] testAllowedRequests_DisallowDecrements() (gas: 28258)
[PASS] testAuthorizeOffer_AfterCancelUsesHigherNonce() (gas: 262258)
[PASS] testAuthorizeOffer_AssetMismatch() (gas: 71349)
[PASS] testAuthorizeOffer_Happy() (gas: 153422)
[PASS] testAuthorizeOffer_NotCaller() (gas: 47047)
[PASS] testAuthorizeOffer_RequestNotAllowed() (gas: 53596)
[PASS] testAuthorizeOffer_RequestNotWhitelisted() (gas: 55738)
[PASS] testAuthorizeOffer_SequentialNonces() (gas: 213327)
[PASS] testAuthorizeOffer_ZeroAmount() (gas: 67525)
[PASS] testAuthorizeOffer_ZeroDuration() (gas: 67530)
[PASS] testBalancesOf() (gas: 34665)
[PASS] testBurn_AssetMismatch() (gas: 269509)
[PASS] testBurn_CannotWithdraw() (gas: 272313)
[PASS] testBurn_ClearsLingeringApproval() (gas: 292524)
[PASS] testBurn_Happy() (gas: 272925)
[PASS] testBurn_NotCaller() (gas: 272067)
[PASS] testBurn_NotRepaid() (gas: 271394)
[PASS] testBurn_RequestWrongFactory() (gas: 270769)
[PASS] testCancelOffer_EmitsOfferCancelled() (gas: 197814)
[PASS] testCancelOffer_Happy_All() (gas: 326940)
[PASS] testCancelOffer_Happy_Partial() (gas: 322151)
[PASS] testCancelOffer_Happy_Single() (gas: 198061)
[PASS] testCancelOffer_NextAuthorizeUsesHigherNonce() (gas: 257436)
[PASS] testCancelOffer_NonceNotIssued() (gas: 151572)
[PASS] testCancelOffer_NonceNotIssued_WhenNoOffersIssued() (gas: 68271)
[PASS] testCancelOffer_NonceTooLow_AlreadyAtOnChain() (gas: 202411)
[PASS] testCancelOffer_NonceTooLow_BelowOnChain() (gas: 233585)
[PASS] testCancelOffer_NotCaller() (gas: 154311)
[PASS] testCancelOffer_RequestNotAllowed() (gas: 154923)
[PASS] testCancelOffer_RequestNotWhitelisted() (gas: 150562)
[PASS] testConstructor() (gas: 2773286)
[PASS] testConstructor_ZeroAsset() (gas: 64612)
[PASS] testConsume_RejectsModifiedAmount() (gas: 211467)
[PASS] testConsume_RejectsModifiedExpectedReturn() (gas: 211425)
[PASS] testConsume_RejectsModifiedExpiration() (gas: 211575)
[PASS] testConsume_RejectsModifiedMaker() (gas: 213556)
[PASS] testConsume_RejectsModifiedNonce() (gas: 211554)
[PASS] testConsume_RejectsModifiedUseCallback() (gas: 211490)
[PASS] testConvertToAssets() (gas: 35644)
[PASS] testCurrentNonce_AfterOnChainAdvances() (gas: 41942)
[PASS] testCurrentNonce_Default() (gas: 21094)
[PASS] testCurrentNonce_PendingOfferCount() (gas: 215045)
[PASS] testDisallowRequest_Happy() (gas: 30481)
[PASS] testDisallowRequest_NotAllowed() (gas: 27930)
[PASS] testDisallowRequest_NotRole() (gas: 23193)
```

```
[PASS] testInitialize_DoubleInit() (gas: 37707)
[PASS] testInitialize_EmptyArraysAllowed() (gas: 3582799)
[PASS] testInitialize_StorageAndRoles() (gas: 69010)
[PASS] testInitialize_ZeroAdmin() (gas: 3381680)
[PASS] testInitialize_ZeroFactory() (gas: 3379806)
[PASS] testInitialize_ZeroHolderInArray() (gas: 3724221)
[PASS] testInitialize_ZeroRoleInArray() (gas: 3578079)
[PASS] testInitialize_ZeroSubvault() (gas: 3379762)
[PASS] testInitialize_ZeroWhitelist() (gas: 3379763)
[PASS] testIsRequestAllowed_False() (gas: 27166)
[PASS] testIsRequestAllowed_True() (gas: 17998)
[PASS] testIsRequestAllowed_Unknown() (gas: 15914)
[PASS] testIsRequestWhitelisted_False() (gas: 24751)
[PASS] testIsRequestWhitelisted_NotWhitelisted() (gas: 32442)
[PASS] testIsRequestWhitelisted_PausedNotWhitelisted() (gas: 37223)
[PASS] testIsRequestWhitelisted_PausedWhitelisted() (gas: 37176)
[PASS] testIsRequestWhitelisted_True() (gas: 31041)
[PASS] testIsRequestWhitelisted_Unknown() (gas: 21328)
[PASS] testIsValidSignature_NoAuthorization() (gas: 16162)
[PASS] testLastIssuedNonce_IncrementsWithEachAuthorize() (gas: 273707)
[PASS] testLastIssuedNonce_InitiallyZero() (gas: 17873)
[PASS] testLastIssuedNonce_ResumesAfterOnChainAdvance() (gas: 229263)
[PASS] testMintAuthorization() (gas: 13665)
[PASS] testNextNonce_OfferConsumedUsesCorrectNonce() (gas: 277892)
[PASS] testOnRequestConsumed_ApprovesAndClearsAuth() (gas: 202595)
[PASS] testOnRequestConsumed_ExceedsPtAuthorization() (gas: 207410)
[PASS] testOnRequestConsumed_InsufficientBalance() (gas: 222767)
[PASS] testOnRequestConsumed_InsufficientYt() (gas: 207463)
[PASS] testOnRequestConsumed_OfferNotAuthorized() (gas: 118183)
[PASS] testOnRequestConsumed_PartialFill_DeletesAuth() (gas: 202574)
[PASS] testOnRequestConsumed_RequestNotAllowed() (gas: 206866)
[PASS] testOnRequestConsumed_RequestNotWhitelisted() (gas: 202526)
[PASS] testPullAssets_Happy() (gas: 102104)
[PASS] testPullAssets_NotSubvault() (gas: 18177)
[PASS] testPull_DoubleConsume_Reverts() (gas: 263498)
[PASS] testPull_Happy() (gas: 264832)
[PASS] testPull_PartialFill() (gas: 258146)
[PASS] testPushAssets_Happy() (gas: 114688)
[PASS] testPushAssets_NotSubvault() (gas: 18133)
[PASS] testPush_DoesNotDuplicateActiveRequest() (gas: 218264)
[PASS] testPush_Happy() (gas: 177337)
[PASS] testPush_InsufficientAuthorization() (gas: 117338)
[PASS] testPush_InsufficientBalance() (gas: 119192)
[PASS] testPush_NotCaller() (gas: 97240)
[PASS] testPush_RequestNotAllowed() (gas: 51914)
[PASS] testPush_RequestNotWhitelisted() (gas: 54127)
[PASS] testPush_SlippageExceeded_ExcessPt() (gas: 248985)
[PASS] testPush_SlippageExceeded_InsufficientYt() (gas: 248976)
Suite result: ok. 100 passed; 0 failed; 0 skipped; finished in 5.39ms (45.21ms CPU time)

Ran 1 test suite in 12.03ms (5.39ms CPU time): 100 tests passed, 0 failed, 0 skipped (100 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 conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

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