
Security Review Report
NM-0843 - XDC Prime Stake



NETHERMIND
SECURITY

(May 08, 2026)

Contents

1	Executive Summary	2
2	Audited Files	3
3	Summary of Issues	3
3.1	Stage 1 Findings	3
3.2	Stage 2 Findings	3
4	System Overview	4
4.1	PrimeStakedXDC_V3	4
4.2	PrimeStakedXDC_V3MigrationBridge	4
4.3	Trust Assumptions and Centralization Risks	5
4.3.1	Migration Manager and Migration Lifecycle	5
4.3.2	Operations Manager	5
4.3.3	Risk Manager	5
4.3.4	Proposer and Masternode Operators	5
5	Risk Rating Methodology	6
6	Issues	7
6.1	[Critical] Returned masternode stakes are double-counted in trackedTotalAssets	7
6.2	[High] Malicious receivers can trigger a gas bomb attack in _processWithdrawalQueueInternal(...) blocking the withdrawal queue	8
6.3	[High] Permissionless syncTrackedAssets() during migration allows early migrators to steal funded liquidity via share price manipulation	9
6.4	[Medium] Deferred withdrawal liabilities are not ring-fenced, enabling share price inflation and claim insolvency	10
6.5	[Low] Flawed _nextOperatorIndex reset logic in removeOperator(...) skips validator turns	11
6.6	[Low] Legacy withdraw() is permanently bricked due to nested nonReentrant guards	12
6.7	[Low] Migration manager can freeze all user-facing operations instantly	12
6.8	[Low] The removeOperator(...) function reshuffles the operators array causing validators to skip their turns	13
6.9	[Low] triggerAutoPropose reverts on validator failure unlike _autoProposeOne	13
6.10	[Info] Bridge withdrawal timelock can be neutralized by instantly changing delay and cap	14
6.11	[Info] Constructor seed without share mint sets an unfavorable initial exchange rate	15
6.12	[Info] No mechanism to resign or unstake masternodes through the contract	15
6.13	[Info] The OperationsManager can pause the protocol via bufferBps manipulation	16
6.14	[Info] Timelocked governance changes can be silently overwritten without cancellation events	17
6.15	[Info] Users can frontrun the reportValidatorLoss(...) function to evade slashing penalties	17
6.16	[Info] V3 has no dedicated emergency pause mechanism	18
6.17	[Info] setValidator and setMinStake emit no events	18
6.18	[Best Practices] Unreachable guard in _syncTrackedAssetsExcludingInflow	19
6.19	[Best Practices] executeOwnerTransfer to self permanently locks admin role	19
6.20	[Low] Constructor does not enforce a non-zero seed at deployment	19
6.21	[Info] redeemWithQueue and withdrawWithQueue use all-or-nothing logic, queueing the entire request even when partial immediate liquidity exists	20
7	Documentation Evaluation	21
8	Test Suite Evaluation	22
8.1	Tests Output	22
8.2	Automated Tools	25
8.2.1	AuditAgent	25
9	About Nethermind	26

1 Executive Summary

This document presents the results of a security review conducted by [Nethermind Security](#) for PrimeNumbers Labs' **PrimeStakedXDC_V3** liquid staking protocol and its accompanying **PrimeStakedXDC_V3MigrationBridge**.

PrimeStakedXDC_V3 is an ERC4626 vault that accepts native XDC and issues psXDC shares in return. Deposited XDC is delegated to XDC Network masternodes through a round-robin operator selection system, and validator rewards flow back to the vault to appreciate the share price over time. The contract provides both instant withdrawals (subject to a configurable liquidity buffer) and a queued withdrawal path for larger exits, with timelocked governance controls for all sensitive parameter changes.

PrimeStakedXDC_V3MigrationBridge facilitates the transition from the existing upgradeable V2 token to the standalone V3 vault. Users migrate by burning their V2 psXDC tokens through the bridge, which mints equivalent V3 shares on their behalf. The underlying XDC backing the V2 supply is separately transferred to the V3 contract by the migration manager to maintain full collateralization.

Section 3.1 (*Stage 1 Findings*) presents the issues identified during the initial audit of the codebase. After the PrimeNumbers Labs team implemented fixes addressing those findings, Nethermind verified the remediations; additional issues that surfaced during that fix verification are presented separately in Section 3.2 (*Stage 2 Findings*).

Along this document, we report 21 points of attention, where one is classified as Critical, two are classified as High, one is classified as Medium six are classified as Low, nine are classified as Informational and two is classified as Best Practices. 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 system overview. Section 4 discusses the risk rating methodology. Section 5 details the issues. Section 6 discusses the documentation provided by the client for this audit. Section 7 presents the test suite evaluation and automated tools used. Section 8 concludes the document.



Fig. 1: Distribution of issues: Critical (1), High (2), Medium (1), Low (6), Undetermined (0), Informational (9), Best Practices (2).
Distribution of status: Fixed (18), Acknowledged (3), Mitigated (0), Unresolved (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	March 26, 2026
Final Report	May 08, 2026
Initial Commit	f92b803
Final Commit	2d97d9b
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	PrimeStakedXDC_V3MigrationBridge.sol	201	6	3.0%	46	253
2	PrimeStakedXDC_V3.sol	1190	19	1.6%	233	1442
	Total	1391	25	1.8%	279	1695

3 Summary of Issues

3.1 Stage 1 Findings

	Finding	Severity	Update
1	Returned masternode stakes are double-counted in trackedTotalAssets	Critical	Fixed
2	Malicious receivers can trigger a gas bomb attack in _processWithdrawalQueueInternal(...) blocking the withdrawal queue	High	Fixed
3	Permissionless syncTrackedAssets() during migration allows early migrators to steal funded liquidity via share price manipulation	High	Fixed
4	Deferred withdrawal liabilities are not ring-fenced, enabling share price inflation and claim insolvency	Medium	Fixed
5	Flawed _nextOperatorIndex reset logic in removeOperator(...) skips validator turns	Low	Fixed
6	Legacy withdraw() is permanently bricked due to nested nonReentrant guards	Low	Fixed
7	Migration manager can freeze all user-facing operations instantly	Low	Fixed
8	The removeOperator(...) function reshuffles the operators array causing validators to skip their turns	Low	Fixed
9	triggerAutoPropose reverts on validator failure unlike _autoProposeOne	Low	Fixed
10	Bridge withdrawal timelock can be neutralized by instantly changing delay and cap	Info	Fixed
11	Constructor seed without share mint sets an unfavorable initial exchange rate	Info	Fixed
12	No mechanism to resign or unstake masternodes through the contract	Info	Fixed
13	The OperationsManager can pause the protocol via bufferBps manipulation	Info	Fixed
14	Timelocked governance changes can be silently overwritten without cancellation events	Info	Fixed
15	Users can frontrun the reportValidatorLoss(...) function to evade slashing penalties	Info	Acknowledged
16	V3 has no dedicated emergency pause mechanism	Info	Fixed
17	setValidator and setMinStake emit no events	Info	Fixed
18	Unreachable guard in _syncTrackedAssetsExcludingInflow	Best Practices	Fixed
19	executeOwnerTransfer to self permanently locks admin role	Best Practices	Fixed

3.2 Stage 2 Findings

	Finding	Severity	Update
20	Constructor does not enforce a non-zero seed at deployment	Low	Acknowledged
21	redeemWithQueue and withdrawWithQueue use all-or-nothing logic, queueing the entire request even when partial immediate liquidity exists	Info	Acknowledged

4 System Overview

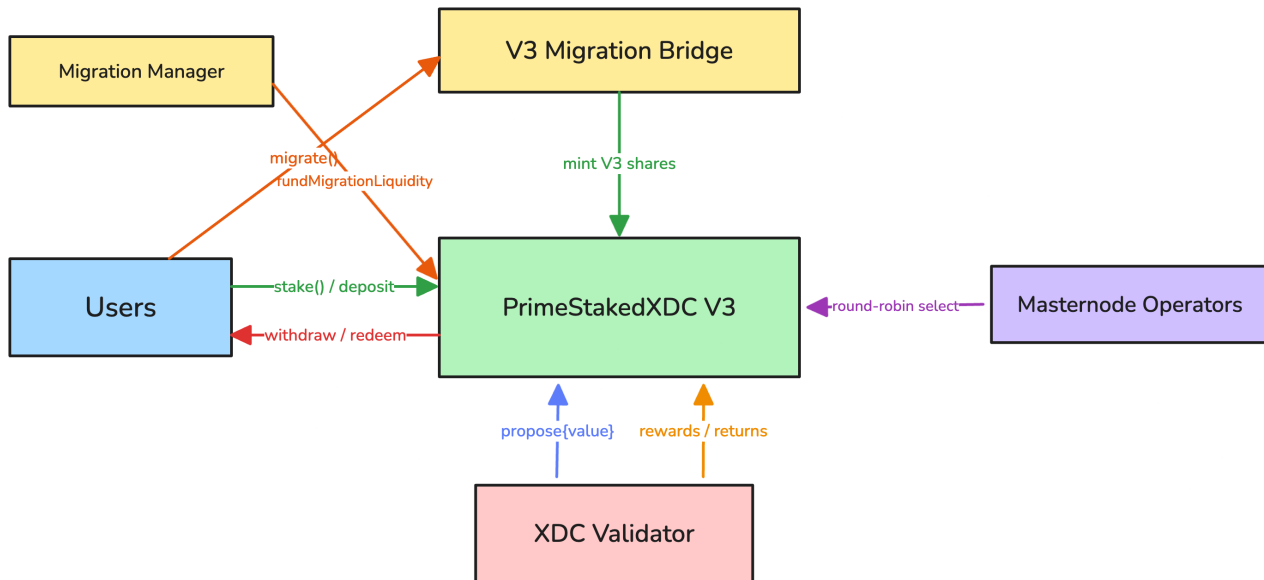


Figure 1: PrimeStakedXDC V3 system architecture and contract interactions.

4.1 PrimeStakedXDC_V3

The **PrimeStakedXDC_V3** contract is a non-upgradeable ERC4626 vault that implements liquid staking for native XDC. Users deposit XDC through `stake()` or `depositNative()` and receive psXDC shares representing their proportional claim on the vault's managed assets. The share price appreciates as validator rewards flow into the contract through its `receive()` function and are incorporated into `trackedTotalAssets`.

Deposited XDC is delegated to XDC Network masternodes via `proposeMasternode()` or through an automatic proposal system, which selects the next eligible operator in a round-robin fashion from a registered operator set. A configurable liquidity buffer (`bufferBps`) is retained in the contract to service instant withdrawals.

The contract provides two withdrawal paths. Instant withdrawals through `withdraw()` and `redeem()` are available up to the liquid buffer. When insufficient liquidity exists, users can queue withdrawals via `redeemWithQueue()` or `withdrawWithQueue()`, which lock their shares in the contract until the queue is processed. Queue processing is permissionless via `processWithdrawalQueue()`, and deposits also opportunistically process one queued request each. If a queued payout transfer fails, the owed amount is recorded in a deferred claims mapping for the user to retrieve later via `claimQueuedAssets()`.

The vault is deployed through a payable constructor that accepts an optional native XDC seed via `msg.value`, which is assigned directly to `trackedTotalAssets`. Notably, no shares are minted at construction time. The contract relies on a `_decimalsOffset` of 9, which introduces virtual shares and virtual assets into the ERC4626 exchange rate formula to mitigate first-depositor inflation attacks. As a result, the effective initial exchange rate is established by the first `migrate()`, `stake()`, or `depositNative()` call, where the ratio of deposited assets to minted shares is governed by the virtual offset rather than a pre-seeded share supply.

Governance follows a timelocked two-step pattern (schedule, then execute after a configurable delay) for all sensitive parameter changes, including role assignments, loss reporting thresholds, and ownership transfers. The contract defines five privileged roles via OpenZeppelin's `AccessControl`: `DEFAULT_ADMIN_ROLE` for governance and role management, `PROPOSER_ROLE` for masternode proposals, `OPERATIONS_MANAGER_ROLE` for day-to-day parameter tuning, `RISK_MANAGER_ROLE` for reporting validator losses, and `MIGRATION_MANAGER_ROLE` for controlling the migration lifecycle and funding liquidity. Direct role mutations through `grantRole()`, `revokeRole()`, and `renounceRole()` are disabled, enforcing that all role changes pass through the timelocked governance flow.

4.2 PrimeStakedXDC_V3MigrationBridge

The **PrimeStakedXDC_V3MigrationBridge** facilitates the one-way migration from the existing upgradeable V2 psXDC token to the standalone V3 vault. Users approve the bridge to spend their V2 tokens and call `migrate()`, which burns the V2 tokens and instructs the V3 contract to mint an equivalent amount of shares to the user. A `minSharesOut` parameter provides slippage protection against exchange rate changes during migration.

Users should claim any accrued V2 staking rewards via `getReward()` before migrating, as the bridge's `transferFrom` call does not trigger the V2 reward accounting for the migrating user. Any unclaimed rewards accrued since the user's last V2 interaction will be forfeited once the migration transfers their token balance to zero.

The migration process is split into two independent streams: share migration (handled by the bridge) and liquidity migration (handled by the migration manager). The migration manager separately transfers the underlying XDC from V2 to V3 via `fundMigrationLiquidity()`, which credits the contract's native balance without altering `trackedTotalAssets` to avoid distorting the share price for subsequent migrators.

The bridge includes a timelocked withdrawal mechanism for recovering any accidentally sent native XDC, with a configurable delay and daily cap. Ownership transfer follows the same two-step timelocked pattern as the V3 contract.

4.3 Trust Assumptions and Centralization Risks

The protocol relies on several trusted roles whose correct and timely behavior is essential for the system to function as intended. The following outlines the key trust dependencies and the risks they introduce.

4.3.1 Migration Manager and Migration Lifecycle

The migration manager controls the `migrationInProgress` flag, which gates all user-facing deposit and withdrawal functions via the `whenMigrationFinished` modifier. While migration is active, the entire protocol is frozen for non-migration operations. The migration manager is also responsible for transferring the underlying XDC from V2 to V3 in the correct amount and at the correct time. If liquidity is funded before all users have migrated, `permissionless` calls to `syncTrackedAssets()` can incorporate migration liquidity into the share price, skewing the exchange rate for later migrators. The protocol assumes the migration manager will fund liquidity only after all share migrations are complete.

4.3.2 Operations Manager

The operations manager controls day-to-day operational parameters without a governance delay: the liquidity buffer target (`bufferBps`), auto-propose configuration, scan limits, the validator oracle address (`validator`), and the minimum masternode stake. The validator address is particularly sensitive, as it determines both the destination for staked XDC and the KYC verification oracle.

4.3.3 Risk Manager

The risk manager reports validator losses via `reportValidatorLoss()`, which directly reduces `trackedTotalAssets` and lowers the share price for all holders. Per-report and daily loss caps bound the magnitude, but within those bounds the risk manager has unilateral authority to reduce the vault's net asset value.

4.3.4 Proposer and Masternode Operators

The proposer role (or admin) controls which operators receive staked XDC and in what amounts. Once XDC is sent to a masternode via `proposeMasternode()`, it cannot be recalled through the contract, as there is no built-in resign mechanism. Registered operators must be KYC-verified through the validator oracle before they can be selected for proposals. The admin manages the operator set via `addOperator()` and `removeOperator()`.

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 [Critical] Returned masternode stakes are double-counted in trackedTotalAssets

File(s): PrimeStakedXDC_V3.sol

Description: When XDC is sent to the validator via proposeMasternode or auto-propose, the native balance decreases but trackedTotalAssets is intentionally left unchanged because the assets are still managed by the protocol, just illiquid at the validator.

```
1 IXDCValidator(validator).propose{value: amount}(candidate);
2 emit MasternodeProposed(candidate, amount);
```

If that staked XDC later returns to the contract (e.g., the masternode is resigned externally, the validator deregisters the node, or a network-level return occurs), the incoming XDC arrives through receive(). The validator contract does not hold any privileged role, so it follows the non-privileged sender path:

```
1 receive() external payable {
2   if (migrationInProgress) {
3     require(
4       hasRole(MIGRATION_MANAGER_ROLE, _msgSender()),
5       UnauthorizedMigrationManager()
6     );
7     emit MigrationLiquidityFunded(_msgSender(), msg.value);
8     return;
9   } else if (
10    !hasRole(OPERATIONS_MANAGER_ROLE, _msgSender()) &&
11    !hasRole(DEFAULT_ADMIN_ROLE, _msgSender())
12  ) {
13    require(msg.value >= MIN_REWARD_DISTRIBUTION, InsufficientLiquidAssets());
14  }
15
16  trackedTotalAssets += msg.value;
17  _syncTrackedAssetsExcludingInflow(0);
18 }
```

The returned stake passes the MIN_REWARD_DISTRIBUTION check and is added to trackedTotalAssets. However, those assets were never removed from trackedTotalAssets when they were originally proposed. The result is that the returned stake is counted twice: once from the original proposal (never deducted) and once from the receive() increment.

The receive() function cannot distinguish between reward inflows (new income that should be tracked) and stake returns (already tracked, should not be added again). This inflates the share price, causing new depositors to receive fewer shares than they should, and allowing existing holders to redeem at an artificially high rate.

Recommendation(s): Consider adding a privileged function that allows the operations manager to receive returned masternode stakes without incrementing trackedTotalAssets, or tracking proposed-but-not-returned amounts so that receive() can offset returns against outstanding proposals.

Status: Fixed

Update from the client: Fixed at commit [e473642](#)

Update from Nethermind: The fix introduced a new bug that leads to yield suppression and wrong trackedTotalAssets accounting.

Update from the client: Fixed at commit [f6b2712](#)

Update from Nethermind: The previous issue was addressed, but the new fix introduced a bug that leads to wrong accounting in the case of slashing events.

Update from the client: Fixed at commit [bf9569b](#)

Update from Nethermind: The main vulnerability resurfaced.

Update from the client: Fixed at commit [86faea6](#)

Update from Nethermind: The issue is partially fixed, but the yield suppression vulnerability resurfaced.

Update from the client: Fixed at commit [2d97d9b](#)

Update from Nethermind: Fixed under the assumption that validators will not return the principal amount staked directly to the contract to trigger the receive() logic. Tokens must always be returned via the resignMasternode(...) flow.

6.2 [High] Malicious receivers can trigger a gas bomb attack in `_processWithdrawalQueueInternal(...)` blocking the withdrawal queue

File(s): contract/PrimeStakedXDC_V3.sol

Description: The PrimeStakedXDC_V3 contract allows the operations manager to process queued user withdrawals via the `processWithdrawalQueue(...)` function. This function iterates through pending requests and attempts to send the owed native XDC to the designated receiver. To gracefully handle receivers that cannot accept native tokens (like certain smart contracts), the protocol catches the boolean result of the external call and defers the payout if it fails.

```

1  function _processWithdrawalQueueInternal(uint256 maxRequests) internal {
2      // ...
3      while (processedCount < maxRequests) {
4          // ...
5          request.processed = true;
6          _burn(address(this), request.shares);
7
8          trackedTotalAssets -= assetsOut;
9          totalWithdrawnByUser[request.owner] += assetsOut;
10         // ...

```

The vulnerability lies in the external call lacking a strict gas limit stipened.

```

1      // ...
2      // @audit-issue External call forwards 63/64 of available gas
3      (bool success, ) = payable(request.receiver).call{value: assetsOut}("");
4      if (!success) {
5          // @audit This remaining logic requires gas to execute successfully
6          pendingQueuedAssets[request.owner] += assetsOut;
7          emit WithdrawalQueuePayoutDeferred(...);
8      }
9
10     emit Withdraw(...);
11     // ...
12 }
13 }

```

An external call forwards 63/64 of the currently available gas to the target address. An attacker can set their withdrawal receiver to a malicious "gas bomb" smart contract. This contract can implement a `receive()` function that purposely runs a loop until it consumes all of its allocated 63/64 gas, eventually reverting the transaction with an Out-of-Gas (OOG) error.

When the call reverts, the EVM returns control to the PrimeStakedXDC_V3 contract with only the remaining 1/64 of the gas. This amount of gas is likely to be insufficient to finish executing the subsequent state updates. Consequently, the parent transaction also reverts due to an OOG error, meaning the `request.processed = true` state change is never saved.

This attack is amplified by the "infinite loop" behavior described in the below issue because it blocks the entire withdrawal queue. Because the failed request is never successfully marked as processed, the `_nextPendingRequestId(...)` function will continuously fetch the exact same malicious request ID on every subsequent attempt. This permanently blocks the entire withdrawal queue for all users. The only way for the operations manager to bypass the malicious request is to pump an exorbitant amount of gas into the transaction, ensuring that the remaining 1/64 gas is enough to complete the loop's execution. However, an attacker can continuously queue withdrawals to grief the protocol, forcing the admin to spend money on gas just to keep the queue operational.

Recommendation(s): Consider enforcing a strict gas limit on the external call to ensure the parent contract always retains enough gas to complete its execution. For example, changing the call to `payable(request.receiver).call{value: assetsOut, gas: 20000}("")` will safely limit the receiver's gas consumption while still allowing standard wallets and simple fallback functions to receive the funds.

Status: Fixed

Update from the client: Fixed at commit [61c2d2e](#)

6.3 [High] Permissionless syncTrackedAssets() during migration allows early migrators to steal funded liquidity via share price manipulation

File(s): PrimeStakedXDC_V3.sol

Description: During migration, native XDC sent by the migration manager through fundMigrationLiquidity() or receive() is intentionally excluded from trackedTotalAssets so that it does not affect ERC4626 share pricing for migrate() calls:

```
1 receive() external payable {
2   if (migrationInProgress) {
3     require(
4       hasRole(MIGRATION_MANAGER_ROLE, _msgSender()),
5       UnauthorizedMigrationManager()
6     );
7     emit MigrationLiquidityFunded(_msgSender(), msg.value);
8     return; // early return, trackedTotalAssets not updated
9   }
10  // ...
11 }
```

```
1 function fundMigrationLiquidity()
2   external
3   payable
4   onlyMigrationManager
5   whenMigrationInProgress
6 {
7   require(msg.value != 0, ZeroStakeAmount());
8   emit MigrationLiquidityFunded(_msgSender(), msg.value);
9   // trackedTotalAssets not updated
10 }
```

However, syncTrackedAssets() is permissionless and has no migration-mode guard. It detects the surplus between address(this).balance and trackedTotalAssets and folds it in as if it were newly arrived reward income:

```
1 function syncTrackedAssets() external {
2   _syncTrackedAssetsExcludingInflow(0);
3 }
```

This allows anyone to call syncTrackedAssets() after migration liquidity has been funded, incorporating that liquidity into trackedTotalAssets mid-migration. Since migrate() computes share amounts via previewDeposit/assets), which depends on totalAssets(), the share pricing for all subsequent migrators changes dramatically.

An attacker who migrates a small amount before the sync receives shares priced against a low trackedTotalAssets. After calling syncTrackedAssets(), all later migrators receive far fewer shares for the same V2 token amount because trackedTotalAssets now includes the full migration liquidity. The attacker ends up holding a disproportionately large fraction of the total share supply and can redeem a corresponding share of the funded liquidity after migration ends.

Recommendation(s): Consider adding a whenMigrationFinished guard to syncTrackedAssets(), or have the sync function account for migration-mode state by skipping the sync when migrationInProgress is true.

Status: Fixed

Update from the client: Fixed at commit [7c169079](#)

6.4 [Medium] Deferred withdrawal liabilities are not ring-fenced, enabling share price inflation and claim insolvency

File(s): PrimeStakedXDC_V3.sol

Description: When a queued withdrawal is processed but the native transfer to receiver fails, the protocol burns the shares and reduces trackedTotalAssets while the XDC remains in the contract balance. The owed amount is recorded in pendingQueuedAssets[owner] for later claiming.

```

1 trackedTotalAssets -= assetsOut;
2 totalWithdrawnByUser[request.owner] += assetsOut;
3
4 (bool success, ) = payable(request.receiver).call{value: assetsOut}("");
5 if (!success) {
6     pendingQueuedAssets[request.owner] += assetsOut;
7 }

```

The deferred payout XDC is not isolated from the rest of the protocol's accounting and liquidity, creating two distinct impacts.

Because the XDC stayed in the contract, address(this).balance now exceeds trackedTotalAssets. Any subsequent call that triggers _syncTrackedAssetsExcludingInflow(0) (which runs inside stake, depositNative, redeem, withdraw, redeemWithQueue, withdrawWithQueue, and syncTrackedAssets) detects this surplus and re-adds it to trackedTotalAssets as if it were newly arrived reward income:

```

1 uint256 currentBalanceExcludingInflow = liquid - inflow;
2 if (currentBalanceExcludingInflow > trackedTotalAssets) {
3     uint256 untracked = currentBalanceExcludingInflow - trackedTotalAssets;
4     uint256 rewardsToSync = (untracked / MIN_REWARD_DISTRIBUTION) * MIN_REWARD_DISTRIBUTION;
5     if (rewardsToSync > 0) {
6         trackedTotalAssets += rewardsToSync;
7     }
8 }

```

When the deferred claimant later calls claimQueuedAssets, the XDC leaves the contract but trackedTotalAssets is not reduced:

```

1 function claimQueuedAssets(
2     address payable receiver
3 ) external nonReentrant returns (uint256 amount) {
4     amount = pendingQueuedAssets[_msgSender()];
5     require(amount != 0, NoQueuedAssetsToClaim());
6     pendingQueuedAssets[_msgSender()] = 0;
7
8     (bool success, ) = receiver.call{value: amount}("");
9     require(success, TransferFailed());
10 }

```

The result is that the same XDC is counted twice: once as backing for outstanding shares (via the re-sync) and once as a deferred liability (via pendingQueuedAssets). This inflates the share price for remaining holders beyond what the contract can actually pay out, creating a solvency gap equal to the deferred amount. This accounting inflation requires the protocol to be in a mostly liquid state where address(this).balance is close to trackedTotalAssets (e.g., early in operation before masternodes are proposed), and requires the deferred amount to be at least MIN_REWARD_DISTRIBUTION (1,000 XDC) for the sync to trigger.

Independently of the sync inflation, the deferred XDC is also not excluded from the protocol's available liquidity. Both _availableForImmediateWithdrawal() and _availableForQueuePayout() use raw address(this).balance without subtracting pending deferred claims:

```

1 function _availableForImmediateWithdrawal() internal view returns (uint256) {
2     uint256 liquid = address(this).balance;
3     uint256 keep = desiredBuffer();
4     if (liquid <= keep) {
5         return 0;
6     }
7     return liquid - keep;
8 }

```

This means other users can consume the deferred XDC through instant withdraw()/redeem() calls, the queue processor can use it to pay other queued withdrawals, and proposeMasternode can lock it into a validator. The victim's shares have already been burned during queue processing, so they cannot re-queue. Their only recovery path is claimQueuedAssets, which reverts if the balance is insufficient, leaving their funds in limbo until fresh deposits or masternode returns replenish the contract. This impact applies even for deferred amounts below MIN_REWARD_DISTRIBUTION and regardless of how liquid the vault is.

Recommendation(s): Consider introducing a global `totalDeferredPayouts` state variable that is incremented when a payout is deferred and decremented in `claimQueuedAssets`. Subtract this value from `address(this).balance` when calculating available liquidity in `_availableForImmediateWithdrawal()` and `_availableForQueuePayout()`. Additionally, subtract it from the balance in `_syncTrackedAssetsExcludingInflow` to prevent deferred amounts from being mistakenly categorized as validator yield. Finally, decrement `trackedTotalAssets` by amount inside `claimQueuedAssets`.

Status: Fixed

Update from the client: Fixed at commit [be88eaf](#)

6.5 [Low] Flawed `_nextOperatorIndex` reset logic in `removeOperator(...)` skips validator turns

File(s): `contracts/PrimeStakedXDC_V3.sol`

Description: The `PrimeStakedXDC_V3` contract distributes stake among registered operators in a round-robin fashion. To select the next eligible operator, the `_selectNextAvailableOperator()` function computes the target index using a modulo operation: `_nextOperatorIndex % _operators.length`.

The problem occurs within the `removeOperator(...)` function. When an operator is removed, the function checks if the `_nextOperatorIndex` is greater than the new `_operators.length`. If it is, the index is reset to 0.

```
1 function removeOperator(address operator_) external onlyAdmin {  
2     // ...  
3     isOperator[operator_] = false;  
4     // @audit-issue Resetting to 0 breaks the ongoing modulo sequence  
5     if (_nextOperatorIndex > _operators.length) {  
6         _nextOperatorIndex = 0;  
7     }  
8  
9     emit OperatorRemoved(operator_);  
10 }
```

By resetting it to 0, the continuous round-robin sequence is destroyed.

For example, imagine a set of 5 validators where the `_nextOperatorIndex` pointer is currently at 9. The next validator to be selected is at index 4 (since $9 \% 5 = 4$). If the admin removes a validator, the array length shrinks to 4. Because $9 > 4$, the reset condition triggers and `_nextOperatorIndex` becomes 0. On the next staking operation, the system will evaluate $0 \% 4 = 0$ and select the validator at index 0.

This leads to an unfair distribution of stakes. The validator previously scheduled for the next turn (index 4) is skipped, while the validator at index 0 is favored.

Recommendation(s): Consider removing the `_nextOperatorIndex` reset logic entirely. Since the `_selectNextAvailableOperator(...)` function relies on modulo arithmetic, the continuously growing index will naturally and correctly wrap around the new array length without requiring a manual reset to 0.

Status: Fixed

Update from the client: Fixed at commit [0fcd1d2](#)

6.6 [Low] Legacy withdraw() is permanently bricked due to nested nonReentrant guards

File(s): PrimeStakedXDC_V3.sol

Description: The no-argument withdraw() function, intended as a legacy-compatible full-balance unstake path for V2 users, carries the nonReentrant modifier and internally calls redeem(), which also carries nonReentrant. OpenZeppelin's ReentrancyGuard uses a shared storage flag that is set to ENTERED on the first call; when the nested redeem() checks the same flag, it finds it already entered and reverts with ReentrancyGuardReentrantCall.

```

1 function withdraw() external nonReentrant returns (uint256 assetsOut) {
2     uint256 shares = balanceOf(_msgSender());
3     assetsOut = redeem(shares, _msgSender(), _msgSender());
4 }

```

```

1 function redeem(
2     uint256 shares,
3     address receiver,
4     address owner
5 )
6 public
7 virtual
8 override
9 nonReentrant
10 whenMigrationFinished
11 returns (uint256 assetsOut)
12 {
13     _syncTrackedAssetsExcludingInflow(0);
14     assetsOut = super.redeem(shares, receiver, owner);
15 }

```

Because withdraw() is an external function calling a public function on the same contract, Solidity treats this as an internal call that shares the same execution context. The reentrancy lock acquired by withdraw() is still held when redeem() attempts to acquire it, causing an unconditional revert on every call regardless of parameters or caller.

This means the legacy withdrawal interface that V2 users would naturally reach for after migration is completely non-functional. Users must instead discover and use redeem(shares, receiver, owner) directly or one of the other withdrawal paths (withdraw(uint256,address,address), redeemWithQueue, withdrawWithQueue).

Recommendation(s): Consider removing the nonReentrant modifier from the no-argument withdraw() since the downstream redeem() call already enforces it.

Status: Fixed

Update from the client: Fixed at commit [20ce84e](#)

6.7 [Low] Migration manager can freeze all user-facing operations instantly

File(s): PrimeStakedXDC_V3.sol

Description: The setMigrationInProgress function allows the migration manager to toggle migration mode with no governance delay:

```

1 function setMigrationInProgress(bool enabled) external onlyMigrationManager {
2     migrationInProgress = enabled;
3     emit MigrationModeUpdated(enabled);
4 }

```

The whenMigrationFinished modifier gates all deposit and withdrawal functions:

- stake(), depositNative() – deposits blocked;
- withdraw(uint256), withdraw(uint256, address, address), redeem(uint256, address, address) – instant withdrawals blocked;
- redeemWithQueue(), withdrawWithQueue() – queued withdrawals blocked;

Setting migrationInProgress = true freezes the entire user-facing protocol.

If the migration manager account is compromised or acts maliciously, the admin cannot override the freeze instantly. The admin must go through the full governance delay cycle to replace the migration manager:

1. Call `setMigrationManager(newAddress)` to start the governance delay timer;
2. Wait for `governanceDelay` to elapse (minimum 1 minute, default 1 day);
3. Call `executeMigrationManager()` to install the new migration manager;
4. Have the new migration manager call `setMigrationInProgress(false)`;

During this entire window, the protocol is completely frozen for all user operations.

Recommendation(s): Consider making the migration toggle a one-time lifecycle: once migration is turned off after being active, it cannot be re-enabled. This removes the ongoing trust assumption on the migration manager after migration completes.

Status: Fixed

Update from the client: Fixed at commit [e221627](#)

6.8 [Low] The `removeOperator(...)` function reshuffles the operators array causing validators to skip their turns

File(s): `contracts/PrimeStakedXDC_V3.sol`

Description: The `PrimeStakedXDC_V3` contract maintains a list of registered operators in the `_operators` array to propose them sequentially for masternode staking. The selection mechanism relies on the `_selectNextAvailableOperator()` function, which uses the `_nextOperatorIndex` state variable to iterate over the array and pick the next eligible operator in a round-robin fashion.

However, the `removeOperator(...)` function does not preserve the original array order when an operator is deleted. To avoid leaving gaps in the array, it swaps the element being removed with the last element in the array and then pops the last element.

```

1  function removeOperator(address operator_) external onlyAdmin {
2      // ...
3      uint256 length = _operators.length;
4      uint256 removeIndex = _operatorIndexPlusOne[operator_] - 1;
5      uint256 lastIndex = length - 1;
6      if (removeIndex != lastIndex) {
7          // @audit-issue This swap moves the last operator to `removeIndex`
8          address lastOperator = _operators[lastIndex];
9          _operators[removeIndex] = lastOperator;
10         _operatorIndexPlusOne[lastOperator] = removeIndex + 1;
11     }
12     _operators.pop();
13     // ...
14 }

```

If an operator positioned at the end of the array is swapped to an index that is lower than the current `_nextOperatorIndex`, that swapped operator will be skipped in the current selection cycle. For example, assume there are 5 operators in the array and `_nextOperatorIndex` is 4 (meaning it's the 5th operator's turn to be selected next). If the admin removes the operator at index 1, the 5th operator (at index 4) is moved to index 1. When the next selection occurs, the cursor will wrap around, meaning the 5th operator lost their turn.

This reshuffling disrupts the fairness of the round-robin distribution.

Recommendation(s): Consider modifying the logic to preserve the order of the operators.

Status: Fixed

Update from the client: Fixed at commit [40bcd8c](#)

6.9 [Low] `triggerAutoPropose` reverts on validator failure unlike `_autoProposeOne`

File(s): `PrimeStakedXDC_V3.sol`

Description: The contract has two internal paths for proposing masternodes. The implicit path used during staking wraps the validator call in try/catch, gracefully handling failures:

```

1  function _autoProposeOne() internal {
2      // ...
3      try
4          IXDCValidator(validator).propose{ value: autoProposeThreshold }(candidate)
5      {
6          emit MasternodeProposed(candidate, autoProposeThreshold);
7      } catch {
8          emit AutoProposeFailed(candidate, autoProposeThreshold);
9      }
10 }

```

The explicit path used by `triggerAutoPropose` does not:

```

1 function _proposeNextMasternode(uint256 amount) internal returns (bool) {
2   address candidate = _selectNextAvailableOperator();
3   if (candidate == address(0) || amount > _availableForImmediateWithdrawal()) {
4     return false;
5   }
6
7   IXDCValidator(validator).propose{ value: amount }(candidate);
8   emit MasternodeProposed(candidate, amount);
9   return true;
10 }

```

If the validator contract reverts on `propose()` for any reason (candidate already registered, validator paused, unexpected state), the entire `triggerAutoPropose` transaction reverts. The implicit stake-path auto-propose continues to work in the same conditions because it catches the error.

Recommendation(s): Consider wrapping the `propose` call in `_proposeNextMasternode` with `try/catch` to match the error-handling behavior of `_autoProposeOne`.

Status: Fixed

Update from the client: Fixed at commit [4747fda](#)

6.10 [Info] Bridge withdrawal timelock can be neutralized by instantly changing delay and cap

File(s): `PrimeStakedXDC_V3MigrationBridge.sol`

Description: The migration bridge implements a time-delayed withdrawal pattern for excess native XDC via `withdrawExcessNative` and `executeExcessNativeWithdrawal`. However, the two protective parameters, `withdrawalDelay` and `dailyWithdrawalCap`, can be changed instantly by the admin with no timelock of their own:

```

1 function setWithdrawalDelay(uint256 newDelay) external onlyAdmin {
2   require(newDelay != 0 && newDelay <= MAX_WITHDRAWAL_DELAY, InvalidDelay());
3   withdrawalDelay = newDelay;
4 }
5
6 function setDailyWithdrawalCap(uint256 amount) external onlyAdmin {
7   require(amount != 0, ZeroAmount());
8   dailyWithdrawalCap = amount;
9 }

```

A compromised admin key can execute the following sequence in a single block: set `withdrawalDelay` to 1 second (the minimum allowed by the `!= 0` check), set `dailyWithdrawalCap` to `type(uint256).max`, schedule a withdrawal via `withdrawExcessNative`, wait 1 second, then call `executeExcessNativeWithdrawal` to drain the entire native balance. This reduces the effective observation window from the intended `DEFAULT_WITHDRAWAL_DELAY` (1 day) to approximately 1 second, defeating the purpose of the time-delayed pattern.

The bridge should hold zero native XDC during normal operation, so the practical impact is limited to accidentally sent XDC. However, the safeguard pattern is misleading as it appears to provide a 1-day observation window when it can be reduced to near-zero.

This is inconsistent with V3, where all analogous sensitive parameter changes go through a governance delay.

Recommendation(s): Consider applying a timelock to changes of `withdrawalDelay` and `dailyWithdrawalCap`, or require that the existing `withdrawalDelay` elapses before the new delay takes effect.

Status: Fixed

Update from the client: Fixed at commit [354fc56](#)

6.11 [Info] Constructor seed without share mint sets an unfavorable initial exchange rate

File(s): PrimeStakedXDC_V3.sol

Description: The V3 constructor is marked payable and assigns msg.value directly to trackedTotalAssets:

```

1  constructor(
2      address initialOwner,
3      address previousPstXDC,
4      address initialProposer,
5      address initialValidator,
6      uint256 initialMinStake
7  ) payable ERC20("PrimeStakedXDC", "psXDC") ERC4626(IERC20(address(0))) {
8      // ...
9      trackedTotalAssets = msg.value;
10     // ...
11 }

```

No shares are minted to match the seeded assets. Because the ERC4626 share-pricing formula (with a _decimalsOffset of 9) computes shares as:

```

1  shares = assets * (totalSupply + 10^9) / (totalAssets + 1)

```

the deployment value of msg.value directly controls the exchange rate seen by the first depositor or migrator and the same rate will be used by subsequent depositors/migrators.

If the contract is deployed with a nonzero msg.value (e.g. 10,000 XDC), totalAssets starts at 10,000e18 while totalSupply remains 0. The first migrator depositing 5,000 XDC would receive:

```

1  shares = 5000e18 * (0 + 10^9) / (10000e18 + 1) = 500,000,000 raw shares

```

With 27-decimal psXDC (18 base + 9 offset), 500,000,000 raw shares is effectively zero display tokens. The migrator's deposit is absorbed into the inflated totalAssets denominator, and the seeded XDC becomes unredeemable dead capital that permanently skews the exchange rate against all subsequent depositors.

Conversely, if the contract is deployed with msg.value = 0, the first migration cleanly establishes a 1:1 peg:

```

1  shares = 5000e18 * 10^9 / (0 + 1) = 5000e27 raw shares = 5,000 psXDC

```

All subsequent deposits then maintain the 1:1 rate until validator rewards begin accruing and naturally shift the exchange rate in favor of existing holders.

Note that the _decimalsOffset of 9 causes V3's decimals() to return 27 (18 base + 9 offset), whereas the live V2 psXDC token uses 18 decimals. While the migration math produces a 1:1 exchange rate in display terms (5000 V2 psXDC becomes 5000 V3 psXDC), the raw balanceOf values differ by a factor of 10⁹. Any off-chain system, wallet integration, or block explorer that hardcodes the V2 decimal count of 18 for the new V3 token will display incorrect values.

Recommendation(s): Consider either deploying with msg.value = 0 so the first migration naturally establishes a 1:1 exchange rate, or minting a proportional number of initial shares to the deployer to match the seeded assets and anchor a safe base rate. If preserving the V2 token's 18-decimal representation is desired, the _decimalsOffset should be set to 0 and the constructor should both seed assets and mint a matching number of shares to establish a safe initial exchange rate that is resistant to vault inflation attacks.

Status: Fixed

Update from the client: Fixed at commit [46ecf38](#)

6.12 [Info] No mechanism to resign or unstake masternodes through the contract

File(s): PrimeStakedXDC_V3.sol

Description: The IXDCValidator interface declares a resign(address _candidate) function, but the V3 contract has no calls to resign and exposes no wrapper for it. Capital flows out to masternodes via proposevalue: amount() but cannot be recalled through the contract:

```

1  // In IXDCValidator.sol
2  function resign(address _candidate) external;

```

```

1  // In PrimeStakedXDC_V3.sol - only propose exists, no resign
2  IXDCValidator(validator).propose{value: amount}(candidate);

```

Recommendation(s): Consider adding a `resignMasternode(address candidate)` function restricted to an appropriate role, calling `IXDCValidator(validator).resign(candidate)`.

Status: Fixed

Update from the client: Fixed at commit [051fd33](#)

6.13 [Info] The OperationsManager can pause the protocol via bufferBps manipulation

File(s): `contracts/PrimeStakedXDC_V3.sol`

Description: The PrimeStakedXDC_V3 contract relies on a liquid buffer to fulfill user withdrawals and process the withdrawal queue. The `setBufferBps(...)` function allows the OperationsManager to adjust this vault's liquidity buffer target.

However, the function allows the OperationsManager to set the bufferBps up to a hardcoded maximum of `10_000`, which represents 100% of the vault's assets.

```
1 function setBufferBps(uint256 newBufferBps) external onlyOperationsManager {
2     // @audit-issue The Operations Manager can set the buffer to 100% (10_000)
3     require(newBufferBps <= 10_000, InvalidThreshold());
4     bufferBps = newBufferBps;
5     emit BufferConfigUpdated(newBufferBps);
6 }
```

If the OperationsManager sets the buffer to `10_000`, the `desiredBuffer()` function will mathematically equal the entirety of the trackedTotalAssets. Consequently, the liquidity checks inside both the `_availableForImmediateWithdrawal(...)` and `_availableForQueuePayout(...)` functions will evaluate to true, as the liquid assets can never legitimately exceed the total tracked assets.

```
1 function _availableForImmediateWithdrawal() internal view returns (uint256) {
2     uint256 liquid = address(this).balance;
3     uint256 keep = desiredBuffer();
4     // @audit When buffer is 100%, keep >= liquid, so this always returns 0
5     if (liquid <= keep) {
6         return 0;
7     }
8     return liquid - keep;
9 }
```

Because these functions will permanently return 0, all withdrawals will revert. Although this role is intended for operational parameter adjustments, this configuration allows the OperationsManager to hold the protocol's TVL blocked. This represents a privilege escalation, bypassing the need for the `DEFAULT_ADMIN_ROLE` to execute a system-wide pause.

Recommendation(s): Consider capping the bufferBps configuration.

Status: Fixed

Update from the client: Fixed at commit [4050280](#)

6.14 [Info] Timelocked governance changes can be silently overwritten without cancellation events

File(s): PrimeStakedXDC_V3.sol, PrimeStakedXDC_V3MigrationBridge.sol

Description: Across both V3 and the migration bridge, timelocked setter functions allow the admin to overwrite a pending change by calling the same setter again without first cancelling the previous one. The overwrite silently replaces the pending value and resets the delay timer, with no cancellation event emitted for the replaced schedule.

In V3, all nine timelocked setters exhibit this pattern: setOperationsManager, setProposer, setRiskManager, setMigrationManager, setMigrationBridge, setMaxLossBpsPerReport, setMaxDailyLossBps, setGovernanceDelay, and scheduleOwnerTransfer. For example:

```

1 function setOperationsManager(address account) external onlyAdmin {
2     require(account != address(0), AddressZero());
3     pendingOperationsManager = account;
4     pendingOperationsManagerExecuteAfter = block.timestamp + governanceDelay;
5
6     emit OperationsManagerChangeScheduled(account, pendingOperationsManagerExecuteAfter);
7 }

```

In the migration bridge, withdrawExcessNative overwrites a pending scheduled withdrawal in the same way:

```

1 function withdrawExcessNative(address payable recipient, uint256 amount) external onlyAdmin {
2     require(recipient != address(0), AddressZero());
3     require(amount != 0, ZeroAmount());
4     require(address(this).balance >= amount, InsufficientNativeBalance());
5
6     pendingWithdrawalRecipient = recipient;
7     pendingWithdrawalAmount = amount;
8     pendingWithdrawalExecuteAfter = block.timestamp + withdrawalDelay;
9
10    emit ExcessNativeWithdrawalScheduled(recipient, amount, pendingWithdrawalExecuteAfter);
11 }

```

Each of these functions has a dedicated cancel* counterpart that properly emits a cancellation event (e.g., OperationsManagerChangeCancelled, ExcessNativeWithdrawalCancelled). When the overwrite path is used instead, off-chain monitoring tools see a new scheduling event but receive no corresponding cancellation event for the previous schedule, creating an audit trail gap.

Recommendation(s): Consider either requiring the existing pending change to be cancelled before scheduling a new one, or emitting a cancellation event inside the setter when overwriting existing pending state.

Status: Fixed

Update from the client: Fixed at commit [361f28b](#)

6.15 [Info] Users can frontrun the reportValidatorLoss(...) function to evade slashing penalties

File(s): contracts/PrimeStakedXDC_V3.sol

Description: The reportValidatorLoss(...) function from the PrimeStakedXDC_V3 contract is utilized by the risk manager to record slashing penalties or losses that occur on the validator side. When called, it reduces the trackedTotalAssets state variable, which subsequently lowers the exchange rate of the psXDC shares for all current depositors.

The contract also provides instant withdrawal capabilities through the withdraw(...) and redeem(...) functions, allowing users to immediately cash out their shares for native assets as long as there is sufficient available liquidity.

```

1 function reportValidatorLoss(uint256 assets) external onlyRiskManager {
2     // ...
3     // @audit-issue Decreasing the assets drops the share price for everyone.
4     trackedTotalAssets -= assets;
5     emit ValidatorLossReported(assets);
6 }

```

```

1  function redeem(
2      uint256 shares,
3      address receiver,
4      address owner
5  )
6      public
7      virtual
8      override
9      nonReentrant
10     whenMigrationFinished
11     returns (uint256 assetsOut)
12 {
13     _syncTrackedAssetsExcludingInflow(0);
14     // @audit Users can instantly exit at the current, pre-slash exchange rate.
15     assetsOut = super.redeem(shares, receiver, owner);
16 }

```

A user or an MEV bot can see the pending `reportValidatorLoss(...)` transaction and frontrun it with a `redeem(...)` or `withdraw(...)` transaction. This allows the user to exit at the pre-slashing exchange rate and avoid the penalty.

Recommendation(s): Consider routing all exit paths through the existing withdrawal queue. Enforcing a time delay between request and execution neutralizes frontrunning.

Status: Acknowledged

Update from the client: Acknowledged

6.16 [Info] V3 has no dedicated emergency pause mechanism

File(s): PrimeStakedXDC_V3.sol

Description: V2 inherited ERC20PausableUpgradeable and provided `pause()/unpause()` functions gated by the owner, allowing all token operations to be halted in an emergency. V3 does not include any pausability. The only flow-control mechanism is `migrationInProgress`, which blocks all user-facing deposit and withdrawal operations.

Since V3 is also non-upgradeable, its code cannot be patched in place if a vulnerability is discovered post-deployment.

Recommendation(s): Consider adding a pausability layer gated by `DEFAULT_ADMIN_ROLE` and applied to core user-facing functions.

Status: Fixed

Update from the client: Fixed at commit [183668b](#)

6.17 [Info] setValidator and setMinStake emit no events

File(s): PrimeStakedXDC_V3.sol

Description: Every other operations manager setter in V3 emits an event upon state change: `setAutoProposeConfig` emits `AutoProposeConfigUpdated`, `setBufferBps` emits `BufferConfigUpdated`, `setOperatorScanLimit` emits `OperatorScanLimitUpdated`, `setQueueScanLimit` emits `QueueScanLimitUpdated`, and `setMigrationInProgress` emits `MigrationModeUpdated`.

However, `setValidator` and `setMinStake` are silent:

```

1  function setValidator(address newValidator) external onlyOperationsManager {
2      require(newValidator != address(0), AddressZero());
3      validator = newValidator;
4  }
5
6  function setMinStake(uint256 newMinStake) external onlyOperationsManager {
7      minStake = newMinStake;
8  }

```

The validator address is particularly sensitive as it controls both where XDC is sent during masternode proposals and the KYC verification oracle (`isKYCVerified` queries `IXDCValidator(validator).getHashCount`). A change to this address without an emitted event creates a monitoring blind spot for off-chain systems, admin dashboards, and governance watchers.

Recommendation(s): Consider emitting events for both `setValidator` and `setMinStake` to maintain consistent observability across all configuration changes.

Status: Fixed

Update from the client: Fixed at commit [609d600](#)

6.18 [Best Practices] Unreachable guard in _syncTrackedAssetsExcludingInflow

File(s): PrimeStakedXDC_V3.sol

Description: The _syncTrackedAssetsExcludingInflow function contains an early return guard checking `liquid < inflow` that can never evaluate to true in any reachable call path.

```

1 function _syncTrackedAssetsExcludingInflow(uint256 inflow) internal {
2     uint256 liquid = address(this).balance;
3     if (liquid < inflow) {
4         return;
5     }
6     // ...
7 }

```

When `inflow` is `msg.value` (the stake and `depositNative` paths), the EVM credits `msg.value` to the contract's balance before the function body executes, so `address(this).balance` always includes `msg.value` and is therefore always greater than or equal to it. No external calls occur between balance crediting and this check. All other callers pass `inflow = 0`, making the comparison `liquid < 0` which is trivially false for a `uint256`.

Recommendation(s): Consider removing the unreachable branch to reduce bytecode size and improve readability.

Status: Fixed

Update from the client: Fixed at commit [218d105](#)

6.19 [Best Practices] executeOwnerTransfer to self permanently locks admin role

File(s): PrimeStakedXDC_V3.sol, PrimeStakedXDC_V3MigrationBridge.sol

Description: In both V3 and the migration bridge, the `executeOwnerTransfer` function grants the admin role to the new owner and then revokes it from the caller:

```

1 function executeOwnerTransfer() external onlyAdmin {
2     address newOwner = pendingOwner;
3     require(newOwner != address(0), NoPendingChange());
4     require(block.timestamp >= pendingOwnerExecuteAfter, ChangeNotReady());
5
6     pendingOwner = address(0);
7     pendingOwnerExecuteAfter = 0;
8
9     _grantRole(DEFAULT_ADMIN_ROLE, newOwner);
10    _revokeRole(DEFAULT_ADMIN_ROLE, _msgSender());
11 }

```

If an admin schedules an ownership transfer to their own address and later executes it, `_grantRole` is a no-op (OpenZeppelin's `_grantRole` returns false when the account already holds the role) while `_revokeRole` succeeds and removes it. Because `grantRole`, `revokeRole`, and `renounceRole` are all overridden to unconditionally revert, the admin role becomes permanently irrecoverable.

The same pattern exists in the bridge. The consequence is a permanent loss of all administrative capabilities on the affected contract, including role management, governance parameter changes, and owner transfers.

Recommendation(s): Consider adding a check that the new owner is not the caller, such as `require(newOwner != _msgSender())`.

Status: Fixed

Update from the client: Fixed at commit [742059d](#)

6.20 [Low] Constructor does not enforce a non-zero seed at deployment

File(s): PrimeStakedXDC_V3.sol

Description: The `PrimeStakedXDC_V3` constructor is payable and seeds `trackedTotalAssets` with `msg.value`, but only mints matching shares when the seed is non-zero. There is no enforcement that the deployer actually provides a seed:

```

1 trackedTotalAssets = msg.value;
2 if (msg.value != 0) {
3     _mint(initialOwner, msg.value);
4 }

```

`_decimalsOffset()` returns 0 (matching V2's 18-decimal `psXDC`), which means the only inflation-attack defense remaining inside `_convertToShares` is OpenZeppelin's +1 virtual offset on both `totalSupply()` and `totalAssets()`. OpenZeppelin's own NatSpec on `ERC4626.sol` (v5.2.0) documents this trade-off: `"While not fully preventing the attack, analysis shows that the default offset (0) makes it non-profitable even if an attacker is able to capture value from multiple user deposits, as a result of the value being captured by the virtual shares (out of the attacker's donation) matching the attacker's expected gains."` OpenZeppelin therefore considers offset 0 an acceptable

choice when developers have other reasons (e.g. preserving a particular `decimals()` value) to use it, but explicitly notes that the attack is not fully prevented.

When the contract is deployed with `msg.value == 0`, the protocol enters a state where `tracked = 0` and `supply = 0`. Until the migration manager calls `setMigrationInProgress(true)`, both `stake()` and the unprivileged `receive()` path remain open (`_syncTrackedAssetsExcludingInflow` only reverts under `whenMigrationFinished`). In that pre-migration window the first depositor establishes the share rate and a malicious first depositor can perform the textbook ERC-4626 inflation pattern: stake 1 wei to mint 1 share, force a donation into `address(this).balance` (either through a `SELFDESTRUCT` of a helper contract or via `receive()` with `msg.value >= MIN_REWARD_DISTRIBUTION`), and let the next state-changing call fold the donation into `trackedTotalAssets`. Subsequent small deposits where $V < D/2 + 1$ then floor to zero shares because `previewDeposit(V) = V * 2 / (D + 2)` rounds down. The victim's native is added to `trackedTotalAssets` but no shares are minted in return.

The practical exploitability of this configuration is limited. Even though the attacker is not directly profitable in the typical case, every victim that does land below the rounding threshold loses their entire deposit, so the configuration is unsafe to ship as a deployment default. The constructor not enforcing a minimum seed is the underlying defect; everything downstream is operational risk.

Recommendation(s): Consider enforcing a non-zero seed at construction time so that `totalSupply()` and `trackedTotalAssets` are bounded below by a meaningful floor from block 0 regardless of deployer behaviour.

Status: Acknowledged

Update from the client: Acknowledged

6.21 [Info] `redeemWithQueue` and `withdrawWithQueue` use all-or-nothing logic, queueing the entire request even when partial immediate liquidity exists

File(s): `PrimeStakedXDC_V3.sol`

Description: `redeemWithQueue` and `withdrawWithQueue` either redeem the full amount immediately or queue the full amount; there is no partial-fill path. When a request exceeds `maxRedeem(owner)` (or `_availableForImmediateWithdrawal()`), the entire request is queued, even if a meaningful fraction could have been satisfied immediately:

```

1  function redeemWithQueue(uint256 shares, address receiver)
2      external nonReentrant
3      returns (bool queued, uint256 requestId, uint256 assetsOut)
4  {
5      require(shares != 0, ZeroShares());
6      _syncTrackedAssetsExcludingInflow(0);
7      address owner = _msgSender();
8
9      if (shares <= maxRedeem(owner)) {
10         assetsOut = super.redeem(shares, receiver, owner);
11         return (false, 0, assetsOut);
12     }
13
14     assetsOut = previewRedeem(shares);
15     requestId = _queueWithdrawal(owner, receiver, shares, assetsOut);
16     return (true, requestId, 0);
17 }
```

If a user holding 100M shares calls `redeemWithQueue(100M, receiver)` while the protocol has 50M XDC of immediate liquidity, `maxRedeem(owner) = 50M`, the check `100M <= 50M` fails, and all 100M shares are escrowed in `address(this)` with `totalPendingQueuedAssets += 100M`. The user receives zero XDC immediately even though 50M was sitting in the contract.

A side effect compounds the cost: `_availableForImmediateWithdrawal()` subtracts `totalPendingQueuedAssets` from `address(this).balance` (clamping to zero when the subtraction would underflow), so after the queue it evaluates to 0 for everyone. Other users can no longer redeem immediately and are pushed into the queue as well, even though 50M of native is still on hand.

A user wishing to combine immediate and queued redemption today must do it manually with two transactions: first call `redeem(maxRedeem(owner), receiver, owner)`, then `redeemWithQueue(remaining, receiver)`. `withdrawWithQueue` exhibits the identical pattern.

Recommendation(s): Consider redesigning `redeemWithQueue` and `withdrawWithQueue` so that the maximum currently-available portion is redeemed immediately and only the residual is enqueued, rather than treating each request as indivisible. This would give users prompt access to whatever liquidity exists, reduce the locked-up `totalPendingQueuedAssets` so other users keep some immediate-withdrawal capacity, and keep the queue head smaller and more likely to fit on the next processing pass.

Status: Acknowledged

Update from the client: Acknowledged

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 PrimeXDC documentation

The documentation for the contracts was provided through the code's Natspec and README files. Moreover, the protocol's development team addressed all questions and concerns raised by the Nethermind Security team during the initial kick-off call and subsequent sync calls and through asynchronous communication, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Tests Output

```
> yarn test

PrimeStakedXDC
  Initialization
    should deploy the contract and set initial state (293ms)
  Staking
    should allow users to stake XDC (2342ms)
    should fail when staking amount doesn't match sent value (2457ms)
    should fail when staking zero amount (2352ms)
  Withdraw Request Flow
    should allow users to request withdrawal
    should allow users to cancel withdrawal request
    should allow owner to approve withdrawal request
    should allow withdrawal after approval
    should revert when non-owner tries to withdraw (1231ms)
    should revert when trying to request withdrawal when already has a request
    should revert when trying to withdraw when user changed balance
  Rewards
    should calculate rewards correctly
    should allow users to claim rewards
    should show correct reward for duration
    should fail if reward amount doesn't match sent value
    should update reward period correctly
  Rewards on a complex scenario
    should handle 3 stakers with different staking durations and rewards (5223ms)
  Owner Withdraw
    should allow owner to withdraw XDC
    should revert when non-owner tries to withdraw
    should revert when trying to withdraw zero amount
    should revert when trying to withdraw more than contract balance
  Reward State Variables
    should track userRewardPerTokenPaid correctly
    should track rewards mapping correctly
    should update lastUpdateTime on reward-affecting actions
    should track rewardPerTokenStored correctly
    should execute exit function correctly
    should mint tokens correctly (825ms)
  Transfer Functions
    should transfer tokens and update rewards for both sender and recipient (1261ms)
    should fail to transfer when paused
    should allow transferFrom with proper approval (1655ms)
    should fail transferFrom without approval (1463ms)
    should fail transferFrom when paused (216ms)
  Migration
    should allow owner to migrate (2673ms)
    should allow migrate and claim rewards (3007ms)
    should revert when no balance tries to migrate (2461ms)
  Pausing
    should allow owner to pause and unpause (3071ms)
  Upgrades
    should allow the owner to upgrade the implementation (1700ms)
    should not allow non-owners to upgrade the implementation (927ms)

PrimeStakedXDC_V3 resign blocker validation
  prevents same-block resignation collision from creating phantom yield (9315ms)
  writes down slashed principal when validator cap drops before withdrawal (6706ms)
```

PrimeStakedXDC_V3 breaker attempts

manual propose credits the requested candidate (5701ms)
 manual propose is not blocked by operator scan-limit ordering (5651ms)
 allows manual propose **for** an operator without operator-level KYC (5462ms)
 reverts manual propose when validator rejects (4819ms)
 treats unsolicited validator transfer as yield **while** principal stays outstanding (5741ms)
 blocks reentrant stake attempt from validator callback (5885ms)
 defers gas-bomb payout and keeps funds claimable (8278ms)
 allows direct redeem **while** an older queued request is still pending (8105ms)
 1) advances head across a large processed dead zone **in** one cancel
 accounts validator inflows as yield outside active resign-withdraw settlement (12083ms)
 processes a live request quickly even with a large processed dead zone (96674ms)
 crosses >512 processed requests and still finds the next live request (686786ms)
 keeps rewards credited and only writes down principal during resign settlement (12844ms)
 allows auto-propose with pending queue when liquid covers queue liabilities and buffer (7680ms)
 recovers auto-propose after canceling a head ahead of a large canceled dead zone (59386ms)
 keeps scan cursor progress across calls and reaches a live tail behind canceled requests (72434ms)
 keeps sub-threshold forced inflow unsynced **until** threshold is reached (3659ms)
 lets a newcomer capture pre-deposit forced rewards once sync threshold is crossed (3384ms)
 does not reprice shares when validator returns zero withdraw cap (8306ms)
 reverts resign settlement when validator does not expose withdraw cap (8758ms)
 keeps validator rewards live during pre-withdrawable resign window (7704ms)
 treats validator inflows as yield even when multiple resign requests are open (11495ms)
 continues to account rewards as yield after subsequent partial resign settlement (11984ms)
 accounts delayed validator transfer as yield **until** explicit withdraw settlement (8322ms)
 allows sync to resume after grace even **if** resign request stays open (8713ms)
 zero-return resign remains open and can be settled later (9643ms)
 cleans up open resign tracking after operator principal is fully written down (8718ms)
 accumulates repeated deferred payouts **for** one owner and clears them **in** one claim (9966ms)
 keeps deferred claims isolated across different owners (10873ms)

PrimeStakedXDC_V3 fork migration

migrates from the three holder accounts on fork (12707ms)
 migrates holders, funds exact liquidity, and completes a real masternode resign flow on fork (159270ms)
 migrates full remaining balances **for** all holders and withdraws exact migrated amounts (8202ms)
 reverts migration when migration mode is disabled (5247ms)
 reverts migration when minted shares are below minSharesOut (4268ms)
 allows owner to schedule and execute excess native withdrawal and blocks non-owner (3336ms)
 reverts owner withdrawExcessNative when amount exceeds bridge balance (2148ms)
 allows owner to cancel a scheduled excess native withdrawal (2464ms)
 requires canceling pending excess native withdrawal before rescheduling (2171ms)
 requires explicit cancel before scheduling a new excess native withdrawal (2283ms)
 enforces daily excess-withdrawal cap and resets next day (4000ms)
 uses delayed bridge owner transfer and blocks direct ownership mutation (2983ms)
 blocks direct V3 ownership mutation and supports delayed owner transfer (2972ms)
 rotates migration bridge and migration manager through delayed governance on fork (2781ms)
 applies operations manager change via delayed governance (2860ms)
 supports canceling pending governance changes on fork (3773ms)
 covers V3 governance validation guards on fork (1398ms)
 covers V3 owner transfer no-pending and cancel paths on fork (2340ms)
 covers bridge validation guards and owner-transfer no-pending paths (2592ms)
 timelocks bridge withdrawal parameter updates (2593ms)
 supports realistic stake/depositNative and immediate redeem/withdraw paths (4825ms)
 updates tracked assets from unexpected native inflows and enforces claim preconditions (2459ms)
 enforces delayed governance **for** risk manager and loss caps on fork (9022ms)
 covers bridge unauthorized owner-only branches and additional withdraw guards (3442ms)

PrimeStakedXDC_V3 randomized invariants

preserves accounting invariants across random actions (seed 7) (24139ms)
 preserves accounting invariants across random actions (seed 42) (21324ms)
 preserves accounting invariants across random actions (seed 1337) (22032ms)
 preserves accounting invariants across random actions (seed 2026) (20364ms)
 preserves accounting invariants across random actions (seed 9001) (20038ms)
 safety property: manual propose should always credit requested candidate (8509ms)
 safety property: delayed validator inflow is yield **until** explicit settle withdraw (11187ms)
 safety property: deferred claim only clears claimant liabilities (11788ms)

PrimeStakedXDC_V3 hardening

- uses conservative default queue scan limit (448ms)
- rejects dust-sized queue withdrawal requests (8307ms)
- blocks stake and withdraw flows during migration mode and resumes after finish (3262ms)
- blocks masternode propose/resign/withdraw flows during migration mode (7989ms)
- allows only migration manager or owner to toggle migration mode (716ms)
- prevents re-enabling migration mode after it has been turned off once (1020ms)
- supports emergency pause and blocks core flows **until** unpaused (3328ms)
- allows only migration manager to top up liquidity during migration and keeps tracked assets unchanged (3173ms)
- blocks syncTrackedAssets during migration so funded liquidity cannot reprice migrators (1227ms)
- defers queue payout when receiver reverts and allows owner to claim later (13032ms)
- does not re-sync deferred queued payouts as rewards (12407ms)
- excludes deferred queued payouts from immediate liquidity and masternode proposals (11680ms)
- processes queued withdrawals using current share value (11096ms)
- caps maxWithdraw and maxRedeem by liquid balance (10515ms)
- includes forced native transfers **in** totalAssets **for** fair pricing (3481ms)
- tracks pure validator reward inflows immediately (9123ms)
- sync excludes queued liabilities **while** including staked principal (10240ms)
- does not increase tracked assets from queue bookkeeping transitions alone (13512ms)
- tracks validator rewards and clears principal via resign withdrawals (12903ms)
- keeps outstanding principal when validator returns only partial resign amount (8501ms)
- allows resign flow to **complete** after a node is 100% slashed (8182ms)
- handles masternode resign proposal and withdrawal flow (7285ms)
- requires operator-aware loss reporting when validator principal is outstanding (11711ms)
- enforces minimum external reward inflow outside migration (1814ms)
- allows owner to report validator-side losses (13926ms)
- reverts invalid validator loss reports (1455ms)
- enforces operations manager role on config updates (2227ms)
- enforces delayed proposer change flow (2090ms)
- activates manager privileges only through delayed execute path (2156ms)
- allows owner to cancel all pending governance changes (3118ms)
- requires canceling pending governance schedules before setting new values (5375ms)
- blocks direct ownership and role mutation functions (743ms)
- supports delayed owner transfer flow (2130ms)
- rejects delayed owner transfer execution when pending owner equals executor (1579ms)
- enforces delayed migration bridge and migration manager changes (3087ms)
- enforces risk manager role and max loss cap (11855ms)
- enforces cumulative daily loss cap (10207ms)
- advances bounded operator scan and still finds a valid operator (14567ms)
- allows auto-propose with queue backlog when liquid covers liabilities and buffer (15827ms)
- does not revert stake when auto-propose fails (4765ms)
- reverts invalid claimQueuedAssets receivers and preserves pending amount on failed claim (11382ms)
- reverts proposeMasternode on unauthorized and invalid stake conditions (5636ms)
- supports resignMasternode with proper authorization and input checks (10445ms)
- emits AutoProposeFailed when validator propose reverts (4955ms)
- covers additional V3 input and state guard branches (7690ms)
- supports withdraw on behalf of owner via allowance (3168ms)
- covers additional governance and config guard thresholds (876ms)
- covers claim and queued cancellation edge branches (10145ms)
- covers immediate paths **for** redeemWithQueue and withdrawWithQueue (2959ms)
- supports legacy withdraw(uint256) full **exit** path (3584ms)
- covers constructor ternary branches **for** proposer and threshold (536ms)
- covers tiny-asset validator-loss cap branches where computed caps round to zero (5678ms)
- covers auto-propose path when operator exists but threshold is not met (5219ms)
- covers withdraw failure with rejecting receiver (6350ms)
- covers constructor guard and initial proposer fallback branches (1234ms)
- accepts native constructor seed and mints matching initial shares (616ms)
- rejects ERC4626 token-style deposit wrappers and reflects KYC view branches (1656ms)
- preserves round-robin fairness after removing an earlier operator (20622ms)
- covers auto-propose precheck branches **for** no validator and missing contract kyc (5381ms)
- covers auto-propose when operators exist but none are kyc-eligible (4697ms)
- covers operator selection fallback branches with no eligible operator (2531ms)
- covers auto-propose **return** when no operator is kyc eligible (2940ms)
- covers native deposit override revert branch (802ms)
- covers queue payout stop on insufficient available and scan-limit branch (12831ms)
- processes a later fillable request when the queue head is temporarily unfillable (12073ms)
- covers unauthorized branches **for** owner-only governance entrypoints (1187ms)
- covers migration governance cancellation and no-pending branches (1987ms)

PrimeStakedXDC_V3 Accounting Tree (Supersized)

Given a healthy protocol with staked assets

Double-counting of validator principal during resignation leads to inflated share prices (8328ms)

Flawed queue accounting locks withdrawals and suppresses validator yield (10161ms)

Single resign request can write down operator-wide outstanding principal (8199ms)

Failed queued payouts artificially inflate `totalPendingQueuedAssets` (8121ms)

Validator losses that exceed `operatorOutstandingPrincipal` are rejected (8028ms)

8.2 Automated Tools

8.2.1 AuditAgent

The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at <https://app.auditagent.nethermind.io>.

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.