# Security Review Report
# NM-0566 WilderWorld zNS Migration

NETHERMIND
SECURITY

(August 15, 2025)

# Contents

# 1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security for WilderWorld zNS migration to zChain. The **zNS** (zero Name Service) is a decentralized domain name system built on the Ethereum blockchain, similar in function to traditional web domains. It supports hierarchical subdomains with no depth limit. To enhance operational efficiency, zNS is being migrated from Ethereum to zChain, a dedicated blockchain.

The migration process consists of three main phases:

a. Freezing the Current System: The existing **zNS** contracts will be upgraded to a pausable version, enabling administrators to halt interactions through external interfaces. This establishes a clear-cut-off point for migration, after which no new domain registration or updates are allowed. The subgraph layer will then synchronize to this snapshot.

b. Data Extraction and Validation: The subgraph will be queried to extract and validate domain and subdomain information. This data will be stored in a MongoDB database for processing.

c. Migration to zChain: Using the data from MongoDB, domain and subdomain registrations will be recreated on zChain, preserving the original hierarchy.

This audit focuses on the zNS migration process across these three phases. In addition, it includes a review of the changes introduced in `zNS v2`, the updated implementation deployed on zChain. Note that the scope of this audit is limited to registration using safe batches; the EOA and JSON-based registration methods are out of scope.

**The audit comprises** 3409 lines of Solidity code and 1722 lines of typescript. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases.

**Along this document, we report** 17 points of attention, where one is classified as `High`, four are classified as `Medium`, five are classified as `Low`, and seven are classified as `Informational` or `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 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.

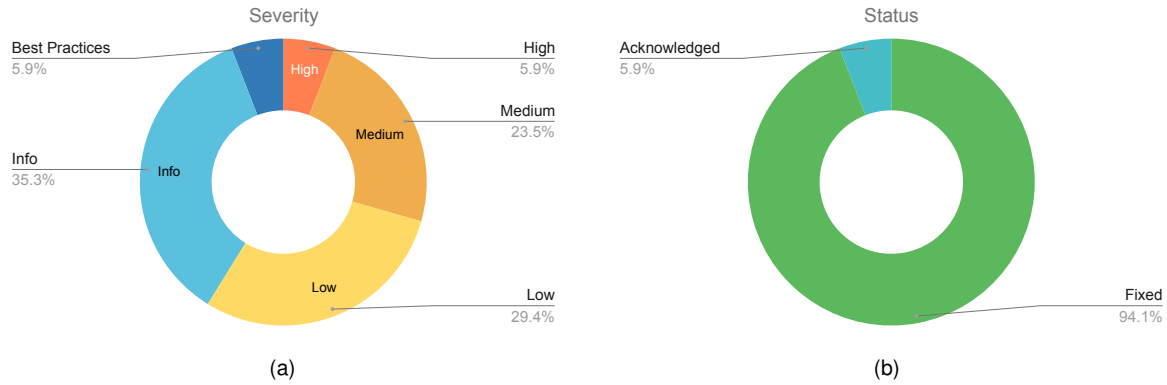


**Fig. 1: Distribution of issues: Critical** (0), **High** (1), **Medium** (4), **Low** (5), **Undetermined** (0), **Informational** (6), **Best Practices** (1). **Distribution of status: Fixed** (16), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

### Summary of the Audit

| | |
|---|---|
| **Audit Type** | Security Review |
| **Initial Report** | July 22, 2025 |
| **Final Report** | August 15, 2025 |
| **Repositories** | zNS |
| **Initial Commit** | **support/v1.0**: 63d094ef71e1f14dec8fb3270d47bbb65c4d4c0b<br>**rc/zchain-native-main**: 4033e82768af737493c3b8c41eb094c6ebbffdc9 |
| **Final Commit** | **support/v1.0**: 2af1b2dc85812d153a96232d00de4f758e876c03<br>**rc/zchain-native-main**: 4711b6ba3a45641fe2cac2b47d6d623ce7db4ea0 |
| **Documentation** | Provided written documentation |
| **Documentation Assessment** | High |
| **Test Suite Assessment** | Medium |

# 2 Audited Files

## 2.1 v1.0 Pausable contracts and Scripts

### 2.1.1 Smart Contracts

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | contracts/zns-pausable/IZNSPausable.sol | 8 | 7 | 87.5% | 6 | 21 |
| 2 | contracts/zns-pausable/resolver/IZNSAddressResolverPausable.sol | 4 | 1 | 25.0% | 3 | 8 |
| 3 | contracts/zns-pausable/resolver/ZNSAddressResolverPausable.sol | 76 | 60 | 78.9% | 19 | 155 |
| 4 | contracts/zns-pausable/registry/ZNSRegistryPausable.sol | 158 | 158 | 100.0% | 40 | 356 |
| 5 | contracts/zns-pausable/registry/ARegistryWiredPausable.sol | 19 | 21 | 110.5% | 8 | 48 |
| 6 | contracts/zns-pausable/registry/IZNSRegistryPausable.sol | 4 | 1 | 25.0% | 3 | 8 |
| 7 | contracts/zns-pausable/registrar/ZNSSubRegistrarPausable.sol | 240 | 115 | 47.9% | 50 | 405 |
| 8 | contracts/zns-pausable/registrar/ZNSRootRegistrarPausable.sol | 238 | 168 | 70.6% | 47 | 453 |
| 9 | contracts/zns-pausable/registrar/IZNSRootRegistrarPausable.sol | 4 | 1 | 25.0% | 3 | 8 |
| 10 | contracts/zns-pausable/registrar/IZNSSubRegistrarPausable.sol | 4 | 1 | 25.0% | 3 | 8 |
| 11 | contracts/zns-pausable/token/ZNSDomainTokenPausable.sol | 161 | 100 | 62.1% | 32 | 293 |
| 12 | contracts/zns-pausable/token/IZNSDomainTokenPausable.sol | 4 | 1 | 25.0% | 3 | 8 |
| 13 | contracts/zns-pausable/price/IZNSFixedPricerPausable.sol | 4 | 1 | 25.0% | 3 | 8 |
| 14 | contracts/zns-pausable/price/ZNSFixedPricerPausable.sol | 111 | 88 | 79.3% | 25 | 224 |
| 15 | contracts/zns-pausable/price/IZNSCurvePricerPausable.sol | 4 | 1 | 25.0% | 3 | 8 |
| 16 | contracts/zns-pausable/price/ZNSCurvePricerPausable.sol | 180 | 170 | 94.4% | 46 | 396 |
| 17 | contracts/zns-pausable/treasury/IZNSTreasuryPausable.sol | 4 | 1 | 25.0% | 3 | 8 |
| 18 | contracts/zns-pausable/treasury/ZNSTreasuryPausable.sol | 198 | 123 | 62.1% | 42 | 363 |
| | **Total** | **1421** | **1018** | **71.6%** | **339** | **2778** |

### 2.1.2 Scripts: Upgrade and Data Acquisition

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | src/upgrade/db.ts | 81 | 1 | 1.2% | 13 | 95 |
| 2 | src/upgrade/upgrade.ts | 104 | 0 | 0.0% | 22 | 126 |
| 3 | src/upgrade/storage-data.ts | 67 | 0 | 0.0% | 13 | 80 |
| 4 | src/upgrade/types.ts | 41 | 0 | 0.0% | 7 | 48 |
| 5 | src/upgrade/scripts/check-paused.ts | 34 | 0 | 0.0% | 5 | 39 |
| 6 | src/upgrade/scripts/execute-upgrade.ts | 29 | 0 | 0.0% | 8 | 37 |
| 7 | src/upgrade/scripts/pause-all.ts | 38 | 0 | 0.0% | 7 | 45 |
| 8 | src/upgrade/scripts/renounce-roles.ts | 40 | 0 | 0.0% | 12 | 52 |
| 9 | src/upgrade/scripts/get-implementations.ts | 24 | 1 | 4.2% | 7 | 32 |
| 10 | src/upgrade/scripts/update-db.ts | 14 | 0 | 0.0% | 5 | 19 |
| 11 | src/utils/migration/database.ts | 34 | 0 | 0.0% | 10 | 44 |
| 12 | src/utils/migration/constants.ts | 3 | 0 | 0.0% | 0 | 3 |
| 13 | src/utils/migration/validate.ts | 79 | 14 | 17.7% | 12 | 105 |
| 14 | src/utils/migration/zns-contract-data.ts | 54 | 2 | 3.7% | 6 | 62 |
| 15 | src/utils/migration/01_validation.ts | 58 | 20 | 34.5% | 21 | 99 |
| 16 | src/utils/migration/types.ts | 81 | 0 | 0.0% | 9 | 90 |
| 17 | src/utils/migration/subgraph/queries.ts | 78 | 0 | 0.0% | 1 | 79 |
| 18 | src/utils/migration/subgraph/index.ts | 33 | 8 | 24.2% | 12 | 53 |
| 19 | src/utils/migration/subgraph/client.ts | 15 | 0 | 0.0% | 5 | 20 |
| | **Total** | **907** | **46** | **0.05%** | **175** | **1128** |

## 2.2 v2.0 Smart contracts and scripts

### 2.2.1 Smart Contracts

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | resolver/IZNSStringResolver.sol | 13 | 7 | 53.8% | 8 | 28 |
| 2 | resolver/ZNSAddressResolver.sol | 55 | 55 | 100.0% | 13 | 123 |
| 3 | resolver/ZNSStringResolver.sol | 53 | 49 | 92.5% | 13 | 115 |
| 4 | resolver/IZNSAddressResolver.sol | 13 | 7 | 53.8% | 8 | 28 |
| 5 | registry/ZNSRegistry.sol | 134 | 166 | 123.9% | 34 | 334 |
| 6 | registry/IZNSRegistry.sol | 76 | 54 | 71.1% | 25 | 155 |
| 7 | registry/ARegistryWired.sol | 18 | 22 | 122.2% | 7 | 47 |
| 8 | registrar/IDistributionConfig.sol | 28 | 41 | 146.4% | 8 | 77 |
| 9 | registrar/ZNSSubRegistrar.sol | 232 | 189 | 81.5% | 52 | 473 |
| 10 | registrar/ARegistrationPause.sol | 23 | 21 | 91.3% | 10 | 54 |
| 11 | registrar/IZNSRootRegistrar.sol | 105 | 96 | 91.4% | 36 | 237 |
| 12 | registrar/IZNSSubRegistrar.sol | 88 | 28 | 31.8% | 28 | 144 |
| 13 | registrar/ZNSRootRegistrar.sol | 243 | 248 | 102.1% | 61 | 552 |
| 14 | token/IZNSDomainToken.sol | 49 | 22 | 44.9% | 18 | 89 |
| 15 | token/ZNSDomainToken.sol | 133 | 146 | 109.8% | 30 | 309 |
| 16 | access/AAccessControlled.sol | 60 | 50 | 83.3% | 11 | 121 |
| 17 | access/IZNSAccessController.sol | 16 | 1 | 6.2% | 14 | 31 |
| 18 | access/ZNSRoles.sol | 9 | 21 | 233.3% | 8 | 38 |
| 19 | access/ZNSAccessController.sol | 58 | 23 | 39.7% | 17 | 98 |
| 20 | access/IZNSRoles.sol | 8 | 2 | 25.0% | 5 | 15 |
| 21 | price/ZNSCurvePricer.sol | 131 | 106 | 80.9% | 31 | 268 |
| 22 | price/IZNSPricer.sol | 22 | 44 | 200.0% | 7 | 73 |
| 23 | price/ZNSFixedPricer.sol | 79 | 58 | 73.4% | 18 | 155 |
| 24 | price/IZNSCurvePricer.sol | 22 | 43 | 195.5% | 11 | 76 |
| 25 | price/IZNSFixedPricer.sol | 16 | 20 | 125.0% | 7 | 43 |
| 26 | oz-proxies/TransparentUpgradeableProxyAcc.sol | 3 | 3 | 100.0% | 3 | 9 |
| 27 | oz-proxies/ERC1967ProxyAcc.sol | 3 | 3 | 100.0% | 3 | 9 |
| 28 | utils/CommonErrors.sol | 8 | 25 | 312.5% | 2 | 35 |
| 29 | utils/StringUtils.sol | 39 | 15 | 38.5% | 7 | 61 |
| 30 | treasury/ZNSTreasury.sol | 166 | 118 | 71.1% | 35 | 319 |
| 31 | treasury/IZNSTreasury.sol | 85 | 66 | 77.6% | 21 | 172 |
| | **Total** | **1988** | **1749** | **88.0%** | **551** | **4288** |

### 2.2.2 Scripts: Registration on zChain

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | database.ts | 33 | 0 | 0.0% | 9 | 42 |
| 2 | constants.ts | 17 | 1 | 5.9% | 3 | 21 |
| 3 | safeKit.ts | 260 | 64 | 24.6% | 48 | 372 |
| 4 | zns-contract-data.ts | 50 | 2 | 4.0% | 5 | 57 |
| 5 | helpers.ts | 199 | 28 | 14.1% | 36 | 263 |
| 6 | 02_registration.ts | 121 | 77 | 63.6% | 32 | 230 |
| 7 | types.ts | 135 | 12 | 8.9% | 17 | 164 |
| | **Total** | **815** | **184** | **22.5%** | **150** | **1149** |

# 3 Summary of Issues

| | Finding | Severity | Status |
|---|---|---|---|
| 1 | Incorrect domain token recipient causes ownership discrepancy after migration | High | Fixed |
| 2 | Duplicate revoked parent causes full batch revert during registration | Medium | Fixed |
| 3 | Incorrect `parent` and `parentHash` assigned to missing parents | Medium | Fixed |
| 4 | Incorrect array used when registering missing revoked parent domains | Medium | Fixed |
| 5 | The Shape of transfers read from MongoDb is incorrect | Medium | Fixed |
| 6 | A Single token owner can block all transfers in batch | Low | Fixed |
| 7 | GraphQL response Errors from Subgraph are not handled | Low | Fixed |
| 8 | Unbounded recursion in `dropDB` can exhaust system resources | Low | Fixed |
| 9 | `compareStorageData` does not support appended state variables in upgraded contracts | Low | Fixed |
| 10 | `createBatchesSafe` may push an empty array to `batchTransferTxs` | Low | Fixed |
| 11 | Admin can replace the Governor role | Info | Fixed |
| 12 | Inconsistent access control for domain token assignment | Info | Acknowledged |
| 13 | Inconsistent priority order for parent hash resolution in subdomain registration | Info | Fixed |
| 14 | Original domain address is not preserved in V2 contract registration | Info | Fixed |
| 15 | Protocol fee can Be deducted from Staked funds | Info | Fixed |
| 16 | Revoked domains remain locked under Safe ownership post migration | Info | Fixed |
| 17 | Improve error handling and validation for `insertMany` on MongoDB collections | Best Practices | Fixed |

# 4 System Overview

The zNS (zero Name Service) protocol is a decentralized domain name management system currently deployed on Ethereum Mainnet. It is being migrated to zChain to improve operational efficiency. To accomplish this, the protocol team is migrating user data from the v1 smart contracts on Ethereum Mainnet to v2 smart contracts on zChain through three main phases:

a. The original v1 system is upgraded to a pausable version, enabling administrators to halt all functionalities and capture a static snapshot of the data.

b. The subgraph is queried to extract domain and subdomain information, which is then validated and stored in a MongoDB database.

c. This database is used to create registration transactions for the v2 contracts on zChain.

It's important to note that the migration includes only domain-related information such as names and ownership. It does not include payment configurations or staked token amounts. The **zNS** team will refund all previous users for their staked amounts, and newly migrated domains will be initialized with zero price and no associated fees. This means domain owners will retain their domains on zChain without needing to lock any tokens in staking mode.

The diagram below summarizes the migration flow for the zNS smart contracts:

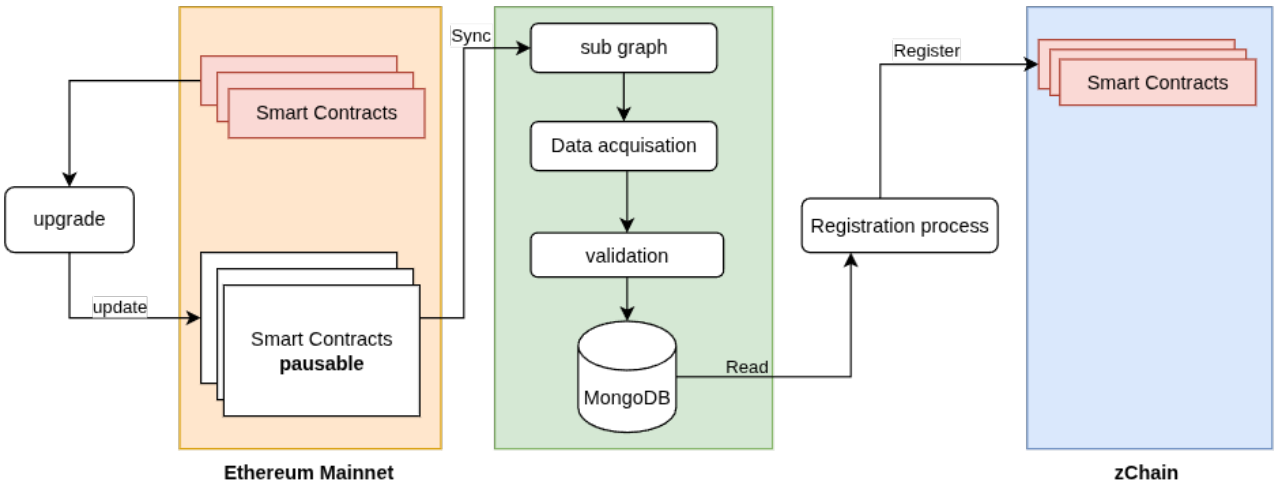


**Fig. 2 Migration Flow Diagram**

## 4.1 zNS Smart Contracts

The zNS protocol relies on a set of core smart contracts to manage domain registration and ownership. The main functionalities have been preserved in the v2 deployment on zChain. Fig. 3 illustrates an overview of the system's key contracts that are described below:

- `zNSRootRegistrar`: Handles the registration of root domains. Anyone can register a unique domain, with pricing determined by the name. Registrants receive ownership and an NFT as proof. Domains can be revoked by their owners, triggering a refund if funds were staked.
- `zNSSubRegistrar`: Enables users to register subdomains under existing domains, forming a hierarchical structure. If a parent domain is revoked, its child domains remain functional and unaffected.
- `zNSRegistry`: Maintains ownership records for domains. Registered owners have administrative rights and can delegate operational permissions across all their domains.
- `zNSDomainToken`: Implements the ERC721 NFT to represent domain ownership. These tokens can be transferred or traded on the market.
- `zNSTreasury`: Manages payments for subdomain creation. When staking is enabled, tokens are locked in this contract and refunded if a domain is revoked.
- `zNSPricing`: Determines pricing for domains and subdomains. Domain owners can configure pricing schemes for subdomains under their control. Two Pricer contract are currently implemented: `zNSFixedPricer` and `zNSCurvedPricer`.

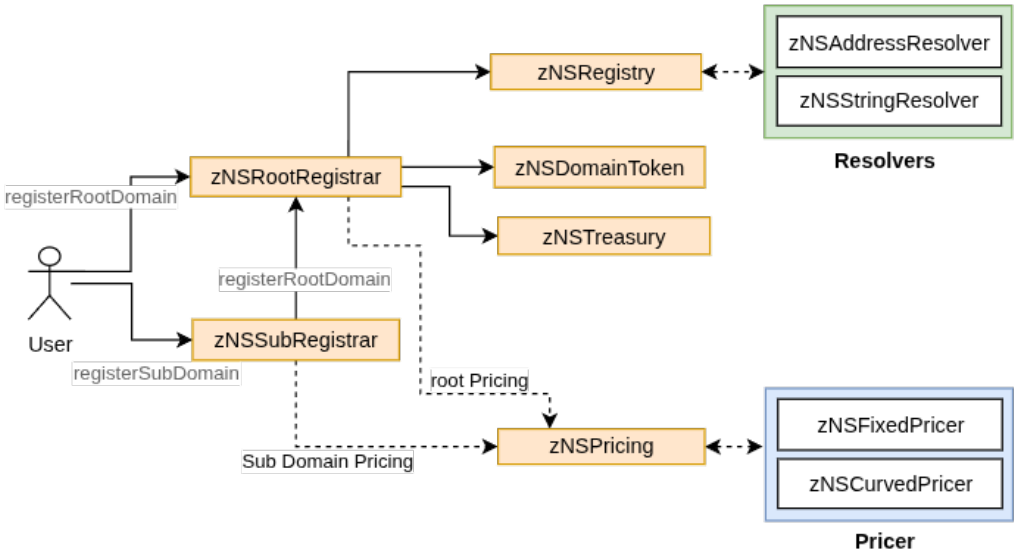


**Fig. 3 zNS Smart Contracts**

## 4.2 Subgraph Layer

A Subgraph layer is maintained to store data from the contracts, including domain and subdomain information. During migration, this layer is used to fetch the final state of the v1 contracts once they are paused, and this data is stored in the database.

## 4.3 Data Acquisition Scripts:

To prepare for migrating `zNS` data from `Ethereum`, data acquisition scripts read information from the `Subgraph` layer, validate the domain hierarchy, and if any links are broken (e.g., due to revoked domains or subdomains), the scripts rebuild the correct hierarchy.

## 4.4 Key changes in v2.0 Smart Contracts

The v2 smart contracts on zChain introduce some differences compared to the v1 contracts on Ethereum Mainnet:

- `Ownership:` In v1, the NFT token owner was considered the main authority. While domain ownership could be delegated in the registry, the token owner could always reclaim control. In v2, primary authority resides with the domain owner, who can assign the domain token to a different address and reclaim it at any time.
- `Pricing:` In v1, each domain could individually define pricing rules across all levels. In v2, pricing is centralized at the root level and managed by the protocol admin. Domain owners can then configure pricing for their direct subdomains.

## 4.5 Registration Process

Once data is collected and the v2 smart contracts are deployed on zChain, the protocol team initiates domain registration using automated scripts. These use a `safe global transaction builder` to batch transactions and ensure atomic execution.

The registration script is executed iteratively to complete different registration steps:

- `Root Domain Registration:` In the initial phase, the script retrieves root domain data from the database and registers them using the `zNSRootRegistrar` contract. These domains are initially assigned to a protocol-owned Safe wallet to retain custody until ownership is properly delegated.
- `Subdomain Registration:` Following the root domain registration, subdomains are registered incrementally by depth. At each depth level, the script checks for missing parent domains (e.g., domains that were revoked on `Ethereum`). If any are found, only the missing parents are registered during that run. The script then halts and is rerun at the same depth level to complete the pending subdomain registrations once the hierarchy is correctly restored.
- `Domain Ownership Transfer:` After all domains (root and subdomains) have been successfully registered, the final step involves transferring ownership and associated NFTs to the rightful v1 owners, as recorded on Ethereum.

It is important to note that the v2 contracts incorporate the `whenRegNotPaused` modifier, which prevents any domain registration by users. This restriction ensures that no user can interact with the v2 registration logic until the migration is fully completed and the protocol explicitly re-enables registration.

Once migration is finalized, domain owners can update their configurations on zChain—including payment methods and distribution settings. Domains remain restricted until owners complete this setup.

## 4.6 Script Configuration and Operational Parameters

The successful execution of the migration process relies on the precise configuration of the migration scripts and their runtime environment. Below are key aspects that must be considered and configured carefully:

- `Duplicate Record Handling:` The input dataset used for registration must be preprocessed to ensure there are no duplicate domain entries at the same hierarchy level. This step is essential to maintain data integrity and prevent conflicting registrations during migration.
- `Depth Control and Hierarchy Management:` The registration script processes the domain hierarchy level by level. Manual specification of the depth parameter is required to control which layer of the hierarchy is processed during each run. Proper tracking of processed depths is essential to ensure complete and consistent domain registration.
- `Environment Variables and Runtime Configuration:` Accurate setup of environment variables such as database access credentials, chain RPC endpoints, contract addresses, and admin keys is critical for successful script execution. Each phase of the migration relies on correctly defined parameters to interact reliably with both the subgraph and smart contracts.

# 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 [High] Incorrect domain token recipient causes ownership discrepancy after migration

**File(s)**: helpers.ts

**Description**: During migration to ZChain, domains are first registered using the `Safe Address` as an intital owner, and afterward, ownership of the domain token (NFT) is transferred to a designated recipient. Currently, the script transfers the token to `domain.owner.id`, which is assumed to be the rightful domain owner.

However, this assumption is incorrect in the context of the protocol's v1 Mainnet deployment. In v1, the token holder has ultimate authority—they can reclaim domain ownership at any time. This differs from v2, where the domain owner is the authoritative entity and can assign or revoke token ownership.

As a result, during migration from v1 to v2, transferring domain tokens to `domain.owner.id` (the v1 domain owner) instead of the actual token holder results in an unfair loss of control. The v1 token holders—who had functional ownership—lose their rights post-migration.

```
const createBatchesSafe = (
  domains : Array<Domain>,
  functionSelector : string,
  registerSliceSize : number,
  transferSliceSize : number
) : CreateBatchesResponse => {
    //...
    if (!domain.isRevoked) {
      const transferEncoding = ZNSDomainToken__factory.createInterface().encodeFunctionData(
        "safeTransferFrom(address,address,uint256)",
          //@audit the domain ownership is set to v1 domain's owner
       [ safeAddress, domain.owner.id, domain.tokenId ]
      );

      transfersBatch.push(transferEncoding);
    }
    //...

}
```

**Recommendation(s)**: Replace `domain.owner.id` with the actual token owner address from the v1 `DomainToken` contract. This ensures that the entity with actual control over the domain in v1 retains that control in v2, maintaining migration integrity.

**Status**: Fixed

**Update from the client**: Fixed in d86e8b8371ae36262b6476dda7b324ba16407c72

## 6.2 [Medium] Duplicate revoked parent causes full batch revert during registration

**File(s)**: 02_registrations.ts

**Description**: When registering subdomains, the script iterates through subdomains of the same depth and checks if their parent domains exist in the registry. If a parent is missing, it is added to a `revokedParents` array so it can be recreated and registered before registering the subdomains beneath it.

However, the current implementation does not check for duplicates in the `revokedParents` array. This might lead to multiple entries for the same parent domain when several subdomains at the same depth share that missing parent. Since domain registration requires each domain to be unique, this results in a revert at the contract level when the same parent is registered more than once in the same Safe batch.

Because the registrations are submitted using Safe's batching mechanism, any revert in the batch causes the entire transaction to fail, including valid entries. This breaks the subdomain registration workflow and blocks progress in the migration or deployment process.

```
const main = async () => {
  //...
  switch (action) {
  case "roots":
    //...
    break;
  case "subs":
    //...
    for (const [i,d] of atDepth.entries()) {
      // ...
      const ownerPromise = zns.registry.getDomainOwner(parentHash);
      const [ owner ] = await Promise.all([ownerPromise]);
      if (owner === ZeroAddress) {
        const missingDomain : Partial<Domain> = {...};

         // @audit `revokedParents` might contain duplicate entries
         revokedParents.push(missingDomain);
      }
    }
  }
    //...
```

**Recommendation(s)**: Ensure that `revokedParents` contains only unique entries before invoking registration on v2 contracts.

**Status**: Fixed

**Update from the client**: Fixed in 7659bb6a8ce22fb35bb097accea2ea8921baeb52

## 6.3 [Medium] Incorrect `parent` and `parentHash` assigned to missing parents

**File(s)**: 02_registrations.ts

**Description**:During the registration process of subdomains on ZChain, the script checks whether a subdomain's immediate parent exists in the registry. If the parent is missing, it is recreated under the `safeAddress` ownership to preserve the hierarchy and enable registration of its subdomains.

To do this, a `Partial<Domain>` object is constructed for the missing parent using available data from the subgraph or fallback values. However, the code mistakenly sets the revoked domain's `parentHash` and `parent.id` fields to its own ID or hash (e.g., `d.parent.id` or `d.parentHash`), rather than referencing its actual parent (i.e., the grandparent of the current subdomain).

As a result, the revoked domain is incorrectly registered as its own parent, leading to invalid hierarchy and broken lineage for any domains beneath it.

```
const missingDomain : Partial<Domain> = {
  label: d.parent?.label,
  owner: {
    id: safeAddress,
    domains: [],
  },
  address: d.parent?.address || safeAddress,
  tokenURI: d.parent?.tokenURI || "http.zero.io",
  tokenId: d.parent?.tokenId,
  // @audit `parentHash` and `parent` will point to the domain itself
  parentHash: d.parent?.parentHash || d.parent?.id,
  parent: {
    id: d.parent?.id,
  },
};
```

**Recommendation(s)**: Update the assignment logic for `parentHash` and `parent.id` to correctly reference the grandparent domain of the subdomain being processed rather than pointing back to the revoked domain itself.

**Status**: Fixed

**Update from the client**: Fixed in bf74e4064c412614ec790997f981dc0f2c308330

## 6.4 [Medium] Incorrect array used when registering missing revoked parent domains

**File(s)**: 02_registrations.ts

**Description**: When registering subdomains, the script must ensure that all parent domains exist in the v2 contract. However, in scenarios where a parent domain has been revoked and is missing from the subgraph, the script attempts to re-register these missing parent domains before proceeding with subdomain registration.

The issue lies in the implementation of this recovery logic: the code mistakenly passes the `atDepth` array (which holds subdomains at the current depth) instead of the correct `revokedParents` array (which contains the parent domains that need to be re-registered).

```
// If there are revoked parents, we propose those instead
   if (revokedParents.length > 0) {
     // We don't catch `transfers` here, we just want these for valid registration
     await proposeRegistrations(
       depth - 1 === 0 ? await zns.rootRegistrar.getAddress() : await zns.subRegistrar.getAddress(),
       safeKit,
       //@audit Errorenous `atDepth` array used instead of `revokedParents`
       atDepth,
       depth - 1 === 0 ? ROOT_DOMAIN_BULK_SELECTOR : SUBDOMAIN_BULK_SELECTOR,
     );

     break;
   }

```

This logic flaw results in the script attempting to re-register subdomains instead of their missing parent domains, leading to a failed execution.

**Recommendation(s)**: Replace the incorrect `atDepth` array with the `revokedParents` array in the `proposeRegistrations` function call.

**Status**: Fixed

**Update from the client**: Fixed in 4071b60e301a5c99145dcd26d7872762c742f497

## 6.5 [Medium] The Shape of transfers read from MongoDb is incorrect

**File(s)**: 02_registration.ts

**Description**: During domain registration, encoded transfer calls are generated to transfer domain tokens from the safe wallet to the rightful domain owners. These calls are grouped into batches and stored in a MongoDB collection with the structure:

```
await client.collection(`${SUB_COLL_NAME}-transfers`).insertOne({ batches: transfers });
```

This process is performed separately for root and subdomain transfers, resulting in two collections. Later, when the script is run in `"transfer"` mode, it reads both collections, merges the data, and passes it to `safeKit.createProposeSignedTxs` for execution.

However, the code incorrectly assumes that calling `.find()` on the collection returns an array of encoded calls. In reality, each document contains a `batches` array nested inside an object. The script currently ignores this structure and attempts to pass the entire object—including the `batches` wrapper—to the Safe execution API.

```
case "transfer":
   // Grab stored transfer txs from earlier runs
   const rootTransfers = client.collection(`${ROOT_COLL_NAME}-transfers`).find() as unknown as Array<string>;
   const subTransfers = client.collection(`${SUB_COLL_NAME}-transfers`).find() as unknown as Array<string>;

   await safeKit.createProposeSignedTxs(
     await zns.domainToken.getAddress(),
     [ ...rootTransfers, ...subTransfers ]
   );
```

As a result, the data passed to `safeKit.createProposeSignedTxs` is malformed, and the transaction execution will fail.

**Recommendation(s)**: The array passed to `createProposeSignedTxs` should be of encoded transfer calls.

**Status**: Fixed

**Update from the client**: Nullified by code refactor. New helper was added that encode transactions before submitting. Fixed in bf74e4064c412614ec790997f981dc0f2c308330

## 6.6 [Low] A Single token owner can block all transfers in batch

**File(s)**: helpers.ts

**Description**: As part of the migration process, domain tokens are transferred from the Safe wallet to their respective owners using `safeTransferFrom`. Each transfer is encoded and collected into a batch of transactions, which is later submitted through the Safe's batch execution mechanism.

```
if (!domain.isRevoked) {
  const transferEncoding = ZNSDomainToken__factory.createInterface().encodeFunctionData(
    "safeTransferFrom(address,address,uint256)",
    [ safeAddress, domain.owner.id, domain.tokenId ]
  );
  transfersBatch.push(transferEncoding);
}
```

The use of `safeTransferFrom` requires the recipient to correctly implement the `IERC721Receiver` interface if the recipient is a smart contract. EOAs can receive tokens without issue, but if any single recipient contract is not deployed yet, or does not implement `onERC721Received`, the transfer to that address will revert.

Since all transfer calls are grouped in a single Safe batch and the current implementation fails to handle single transfer failures, a single invalid recipient will cause the entire batch to fail, preventing successful transfers to otherwise valid owners.

**Recommendation(s)**: Ensure that individual transfer failures do not block the entire batch. Failed transfers can be tracked separately in the database and retried or handled in isolation.

**Status**: Fixed

**Update from the client**: Fixed in b12666122b68fc65a0cf81141ba72a13cb815a83

## 6.7 [Low] GraphQL response Errors from Subgraph are not handled

**File(s)**: index.ts

**Description**: When using Apollo Client to query the Subgraph, the response includes two key fields: `data` and `errors`. While `data` contains the returned results, any issues with the query (e.g. partial failures, invalid fields, or server errors) are captured in the `errors` array.

In the current implementation, the logic only checks for a non-existent `result.error` field, which will never be populated by Apollo. As a result, even if the GraphQL query fails or returns partial results, the application silently continues without noticing the failure. Failing to validate the `errors` array after each query can result in silent data loss or inconsistencies.

```
export const getDomains = async (isWorld : boolean) => {
  // ...
  while (result.data.domains.length > 0) {
    //...

    result = await client.query({
      query: q.getDomains,
      variables: {
        first,
        skip,
        isWorld,
      },
    });
    // @audit no check for `errors` in the result or in the while conditions
  }
  return domains;
};
```

**Recommendation(s)**: Check the returned `errors` field to ensure there is no error in Subgraph data.

**Status**: Fixed

**Update from the client**: Fixed in 05aaae9c24019e49de11d314caa7bcb0c2a49894

## 6.8 [Low] Unbounded recursion in `dropDB` can exhaust system resources

**File(s)**: drop-db.ts

**Description**: In the `dropDB` function, if an error occurs while connecting to the database, the error handler attempts to start MongoDB and recursively calls `dropDB` again. This retry logic lacks a termination condition or a retry limit.

If the failure is persistent, the recursion continues indefinitely, potentially leading to a stack overflow or resource exhaustion.

```
catch (e) {
  // @audit log the exception
  await startMongo();
  await dropDB(); // @audit infinite recursion
}
```

**Recommendation(s)**: Avoid unbounded recursive retries. Log the error and allow for controlled handling or termination.

**Status**: Fixed

**Update from the client**: Fixed in aab37e7bcdbb7d7742d55e85f08bc5d37369931c

## 6.9 [Low] `compareStorageData` does not support appended state variables in upgraded contracts

**Files(s)**: `storage-date.ts`

**Description**: The `compareStorageData` function currently assumes that the `dataBefore` and `dataAfter` arrays have the same length and structure. However, in the current upgrade, the contract's storage was modified by appending a new state variable ( `paused`). This change is not accounted for in the function's logic.

The function uses `.reduce()` to iterate over `dataAfter`, and at each step, it directly accesses `dataBefore[idx]`. When `idx` exceeds the bounds of `dataBefore` (i.e., `dataAfter` includes new variables beyond the original storage layout), the expression `dataBefore[idx]` evaluates to `undefined`. As a result, attempting to access `dataBefore[idx][key]` throws a runtime error, even though the upgrade itself is valid.

This makes the function prone to false negatives in legitimate upgrade scenarios where variables are safely added at the end of the contract.

```
export const compareStorageData = (
  dataBefore : ContractStorageData,
  dataAfter : ContractStorageData,
) => {
  const storageDiff = dataAfter.reduce(
    (acc : ContractStorageDiff | undefined, stateVar, idx) => {
      const [key, value] = Object.entries(stateVar)[0];
      // @audit dataBefore [last index] is undefined
      if (value !== dataBefore[idx][key]) {
        console.error(
          `Mismatch on state var ${key} at idx ${idx}! Prev value: ${dataBefore[idx][key]}, new value: ${value}`
        );

        ...
      }
    }, []
  );

  if (storageDiff && storageDiff.length > 0) {
    throw new Error(`Storage data mismatch: ${JSON.stringify(storageDiff)}`);
  }
};

```

**Recommendation(s)**: Update the function logic to take into consideration the added storage variable in the upgraded contracts.

**Status**: Fixed

**Update from the client**: Fixed in e5acb457ec2e62effb55952a6c517c5c70140025

## 6.10 [Low] `createBatchesSafe` may push an empty array to `batchTransferTxs`

**File(s)**: helpers.ts

**Description**: In the `createBatchesSafe` function, the `transfersBatch` array accumulates encoded transfer instructions, which are pushed to `batchTransferTxs` once a defined `transferSliceSize` is reached.

However, after pushing a full batch and resetting it ( `transfersBatch`), the second conditional will still evaluate to `true` on the last iteration, even if `transfersBatch` is empty, due to the `index + 1 === domains.length` condition.

This results in `batchTransferTxs` including an unnecessary empty array, which could cause the transaction to revert.

```
    if (!domain.isRevoked) {
      const transferEncoding = ZNSDomainToken__factory.createInterface().encodeFunctionData(
        "safeTransferFrom(address,address,uint256)",
        [ safeAddress, domain.owner.id, domain.tokenId ]
      );

      transfersBatch.push(transferEncoding);
    }

    if (transfersBatch.length === transferSliceSize) {
      batchTransferTxs.push(transfersBatch);
      transfersBatch = []; // reset transfers
    }
      //@audit In cases where the domains length  is divisible by the slice size, an empty array is pushed at the end of
       ↪  batchTransferTxs

      if (transfersBatch.length === transferSliceSize || index + 1 === domains.length) {
      // At the last set, so regardless of how much we have we push
      batchTransferTxs.push(transfersBatch);
    }
```

**Recommendation(s)**: Consolidate the two conditional blocks and remove the unnecessary duplicated checks.

**Status**: Fixed

**Update from the client**: Fixed in bf74e4064c412614ec790997f981dc0f2c308330

## 6.11 [Info] Admin can replace the Governor role

**File(s)**: AAccessControlled.sol

**Description**: In the current implementation, the `_setAccessController` function allows an address with the `ADMIN_ROLE` in the current `accessController` contract to update the controller address. This check ensures only an admin can perform the update, but it introduces a governance bypass.

Since the `accessController` defines and manages roles—including the highest authority, `GOVERNOR_ROLE`—allowing an admin to swap out this contract means the admin can effectively assign themselves governor-level privileges. This breaks the intended trust hierarchy by enabling a lower-privileged role (admin) to override the highest one (governor).

**Recommendation(s)**: Reassess the role hierarchy implemented within the system to ensure that critical administrative actions cannot be performed by lower-privileged roles.

**Status**: Fixed

**Update from the client**: Fixed in da3563a9103ca59f9623ff8f2163c2c28d425871

## 6.12 [Info] Inconsistent access control for domain token assignment

**File(s)**: ZNSRootRegistrar.sol

**Description**: The `assignDomainToken` function is designed to reassign a domain token to a new address without changing the domain ownership. Importantly, this function restricts execution solely to the domain owner, explicitly disallowing operators from assigning domain tokens.

However, there is a bypass of this restriction when creating subdomains via the `registerSubdomain` function. During subdomain registration, the operator is allowed to specify an arbitrary `tokenOwner` in the `CoreRegisterArgs` struct. This effectively enables them to assign the domain token to any address, even though the same action would be disallowed in other contexts due to permission checks in `assignDomainToken`.

This discrepancy creates an authorization inconsistency: operators are restricted in one function but inadvertently granted similar authority in another.

```
function registerSubdomain(...) public override whenRegNotPaused(accessController) returns (bytes32) {
    // ...

    CoreRegisterArgs memory coreRegisterArgs = CoreRegisterArgs({
        parentHash: args.parentHash,
        domainHash: domainHash,
        label: args.label,
        domainOwner: domainRecordOwner,
        //@audit this gives the operator the right to assign the domain token to any address
        tokenOwner: args.tokenOwner == address(0) ? domainRecordOwner : args.tokenOwner,
        price: 0,
        stakeFee: 0,
        domainAddress: args.domainAddress,
        tokenURI: args.tokenURI,
        isStakePayment: parentConfig.paymentType == PaymentType.STAKE,
        paymentConfig: args.paymentConfig
    });

    // ...
}

```

**Recommendation(s)**: Consider aligning the authorization model of `registerSubdomain` with that of `assignDomainToken` to ensure consistent access control.

**Status**: Acknowledged

**Update from the client**: We consider this a part of the feature that allows domain operators to assign tokens directly to the users in the case of controlled domains (sending token to address different from canonical/hash owner in Registry). Since the parent domain owner is always assigned as the owner of subdomain registered by an operator, wrong or malicious token assignment here does not pose any risk and could be easily recovered if needed. Token owner can not manage domains or any of the data, only use them.

## 6.13 [Info] Inconsistent priority order for parent hash resolution in subdomain registration

**File(s)**: helpers.ts, 02_registration.ts

**Description**: When registering subdomains, the scripts derive the `parentHash` using two available sources from subgraph data: `domain.parentHash` and `domain.parent.id`. However, the order of precedence between these two fields is inconsistent between `helpers.ts` and `02_registration.ts`.

In `helpers.ts`, the script prioritizes `domain.parentHash` over `domain.parent.id`:

```
const createBatchesSafe = (...) : CreateBatchesResponse => {
  // ...
  for (const [index, domain] of domains.entries()) {
                // ...
    if (functionSelector === SUBDOMAIN_BULK_SELECTOR) {
  // Subdomain, check parentHash
  let parentHash;
    //@audit `parentHash` is prioritized over `parent.id`
  if (domain.parentHash && domain.parentHash !== ZeroHash) {
    parentHash = domain.parentHash;
  } else if (domain.parent?.id && domain.parent?.id !== ZeroHash) {
    parentHash = domain.parent?.id;
  } else {
    throw Error("No parentHash for subdomain. Registration requires parent hash to be set");
  }
   // ...
   }

  // ...
};
```

In contrast, `02_registration.ts` gives priority to `domain.parent.id` over `domain.parentHash`:

```
    for (const [i,d] of atDepth.entries()) {
      let parentHash;
      if (d.parent && d.parent.id) {
        parentHash = d.parent.id;
      } else if (d.parentHash) {
        parentHash = d.parentHash;
      } else {
        // Neither value is readable
        throw Error(`No parent information found for subdomain at ${i}: ${d.label}, ${d.id}`);
      }
```

To ensure consistency between the two scripts, use the same order when considering the parent hash fields.

**Recommendation(s)**: Align the logic across both `helpers.ts` and `registration.ts` by enforcing a consistent order of precedence when resolving the `parentHash`.

**Status**: Fixed

**Update from the client**: Fixed 1eb5fc711d59b9ed4c1b31d4fee06d4becf50a03

## 6.14 [Info] Original domain address is not preserved in V2 contract registration

**File(s)**: helpers.ts

**Description**: In the helpers.ts script, during the creation of domain registration batches, the domainAddress field is set to the domain owner's address (e.g., the actual owner or a Safe address in case of revoked parents). This effectively overwrites the original domain address, which was recorded in the v1 contracts. As a result, when domains are registered in the v2 contracts, the original domainAddress is not recorded on-chain.

```
const createBatchesSafe = (
  domains : Array<Domain>,
  functionSelector : string,
  registerSliceSize : number,
  transferSliceSize : number
) : CreateBatchesResponse => {
  // ...
  for (const [index, domain] of domains.entries()) {
    const args = {
      name: domain.label,
      domainAddress: domain.owner.id, //@audit domain address is replaced by the domain owner's address ( real owner or
      ↪  safe for revoked parents case)
      tokenOwner: process.env.SAFE_ADDRESS!,
      tokenURI: domain.tokenURI,
      distrConfig: {
        pricerContract: ZeroAddress,
        paymentType: 0n,
        accessType: 0n,
        priceConfig: "0x",
      },
      paymentConfig: {
        token: ZeroAddress,
        beneficiary: ZeroAddress,
      },
    };

   // ...
};
```

**Recommendation(s)**: Revisit the intended behaviour and update the domainAddress field accordingly.

**Status**: Fixed

**Update from the client**: Fixed in 1c524aaa75129ede75e6bdd9764a5b82f37717a3

## 6.15 [Info] Protocol fee can Be deducted from Staked funds

**File(s)**: ZNSTreasury.sol

**Description**: In the `unstakeForDomain` function, the current code expects the domain owner to pay the protocol fee separately when unstaking. However, since the staked tokens are already held by the contract, the fee could be taken directly from those funds.

Currently, the contract tries to transfer the fee from the user's wallet. This means the user must have extra tokens and give approval again, even though the fee could be taken from the staked amount.

```
function unstakeForDomain(
        bytes32 domainHash,
        address owner,
        uint256 protocolFee
    ) external override onlyRegistrar {
        Stake memory stakeData = stakedForDomain[domainHash];
        delete stakedForDomain[domainHash];
        //@audit `protocolFee` can be deducted from `stakeData.amount`
        if (protocolFee > 0) {
            stakeData.token.safeTransferFrom(
                owner,
                paymentConfigs[0x0].beneficiary,
                protocolFee
            );
        }

        stakeData.token.safeTransfer(owner, stakeData.amount);

        emit StakeWithdrawn(
            domainHash,
            owner,
            address(stakeData.token),
            stakeData.amount
        );
}
```

**Recommendation(s)**: Consider taking the protocol fee from the staked tokens, and returning the remaining amount to the user.

**Status**: Fixed

**Update from the client**: Fixed in 01ff4560757c2e8e4d2232e788388f15128855b3

## 6.16 [Info] Revoked domains remain locked under Safe ownership post migration

**File(s)**: `helpers.ts`

**Description**: During migration, parent domains that were revoked in v1 are automatically recreated under the Safe address to enable subdomain registration. This workaround is necessary because without a valid parent, subdomain registration transactions would revert.

However, these recreated domains are not marked as revoked in v2 and remain under Safe ownership, making them appear as active domains. This behavior diverges from v1, where revoked domains were permanently deallocated and could later be re-registered by users.

As a result, the domain names cannot be reused or reclaimed by users in v2, nor can new subdomains be created under them. This creates a lock-in effect where placeholder domains remain indefinitely owned by the Safe.

**Recommendation(s)**: Consider revoking all Safe-owned domains created solely for migration purposes, to align with the v1 behavior and ensure they do not remain indefinitely locked in v2.

**Status**: Fixed

**Update from the client**: New code and scripts added for revocations in 48fb12d1498f1bc7466fd2c8afedd91d262cc84a

## 6.17 [Best Practice] Improve error handling and validation for `insertMany` on MongoDB collections

**File(s)**: 01_validation.ts

**Description**: The current implementation uses MongoDB's `insertMany` method to perform bulk inserts into the `validRoots` and `validSubs` collections. However, it does not include any error handling or verification of the insert results.

This is particularly important in migration workflows, where data integrity is critical. The `insertMany` operation returns an object containing an `insertedCount` field, which should be checked to ensure that all records were inserted as expected.

```
const main = async () => {
  //....
  // To avoid duplicate data, we clear the DB before any inserts
  await client.dropCollection(ROOT_COLL_NAME);
  await client.collection(ROOT_COLL_NAME).insertMany(validRoots);

  await client.dropCollection(SUB_COLL_NAME);
  await client.collection(SUB_COLL_NAME).insertMany(validSubs);

  //...
};
```

**Recommendation(s)**: Add appropriate error handling and verify that the number of inserted records matches the number of inputs to ensure data consistency during migrations.

**Status**: Fixed

**Update from the client**: Fixed in fbd93d5d00e57d9ff8231ec5ce0c428c5abe4150

# 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 Wilderworld zNS documentation**

The `Wilderworld` team provided comprehensive technical documentation covering all code within the audit's scope, including a clear breakdown of the migration steps. During the kick-off call, they presented a thorough overview of the migration process and explained the intended functionalities in detail. Moreover, the team addressed all questions and concerns raised by the `Nethermind Security` team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

# 8 Test Suite Evaluation

### 8.0.1 Compilation and Tests Output

```
> yarn test
yarn run v1.22.19
warning ../../../../../package.json: No license field
$ yarn mongo:start
warning ../../../../../package.json: No license field
$ docker-compose up -d
zns_mongo_1 is up-to-date
$ hardhat test

  Controlled Domains Test
Warning: All subsequent Upgrades warnings will be silenced.

    Make sure you have manually checked all uses of unsafe flags.

    Registration
       should register a controlled subdomain as an owner of parent domain by setting token owner to another address
      should register a subdomain as an operator of parent domain but assign parent owner as registry owner of subdomain
       should revert when trying to register a subdomain as anyone other than owner or operator of parent
       should register a controlled root domain by assigning token owner to a separate address
    Domain Management
       should NOT let Root Domain token owner access domain management functions
       should allow registry owner or operator of a controlled domain mint its subdomains
       should NOT let Subdomain token owner access domain management functions
       should allow registry owner or operator of a controlled domain mint its subdomains
      Root Domain Token Rights and Transfers
         should NOT allow controlled domain owner (token owner only) to transfer the token
         should NOT allow approved spender to transfer the controlled domain token
        should allow registry owner to reassign ownership of the controlled domain token back to himself or anyone else
         should allow approved spender to transfer the subdomain token if owner is unified
      Root Domain Revocation
         should NOT allow domain token owner to revoke his domain
         should allow domain owner in registry to revoke subdomain
      Subdomain Token Rights and Transfers
         should NOT allow controlled domain owner (token owner only) to transfer the token
         should NOT allow approved spender to transfer the controlled domain token
        should allow registry owner to reassign ownership of the controlled domain token back to himself or anyone else
         should allow approved spender to transfer the subdomain token if owner is unified
      Subdomain Revocation
         should NOT allow domain token owner to revoke his domain
         should allow domain owner in registry to revoke subdomain

  zNS + zDC Single Integration Test
     Successfully mints TLDs with varying length (39ms)
     Mints subdomains with varying length for free as the owner of parent domain
     Mints subdomains with varying length for a cost (53ms)
     Revokes a domain correctly
     Reclaims a domain correctly by assigning token back to hash owner
     Revokes the domain correctly

  Deploy Campaign Test
    Deploy
       should pause registration on both Registrars if PAUSE_REGISTRATION env variable is set to true (116ms)
       should set `rootPaymentType` correctly based on the ENV var (235ms)
       should set `rootPricerType` correctly based on the ENV var (244ms)
    MEOW Token Ops
       should deploy new ERC20Mock when `mockMeowToken` is true (241ms)
       should use existing deployed MeowToken contract when `mockMeowToken` is false (252ms)
    Failure Recovery
       [in AddressResolver.deploy() hook] should ONLY deploy undeployed contracts in the run following a failed run
       ↪  (432ms)
       [in AddressResolver.postDeploy() hook] should start from post deploy sequence that failed on the previous run
       ↪  (437ms)
      [in RootRegistrar.deploy() hook] should ONLY deploy undeployed contracts in the run following a failed run (529ms)
       [in RootRegistrar.postDeploy() hook] should start from post deploy sequence that failed on the previous run
       ↪  (544ms)
```

```
  Configurable Environment & Validation
    Gets the default configuration correctly
    Confirms encoding functionality works for env variables
    Modifies config to use a random account as the deployer (333ms)
    Fails when governor or admin addresses are given wrong
    Throws if env variable is invalid
    Fails to validate when mocking MEOW on prod
    Fails to validate if not using the MEOW token on prod
    Fails to validate if no mongo uri or local URI in prod
  Versioning
    should get the correct git tag + commit hash and write to DB (78ms)
    should create new DB version and KEEP old data if ARCHIVE is true and no TEMP versions currently exist (316ms)
    should create new DB version and WIPE all existing data if ARCHIVE is false and no TEMP versions currently exist
    ↪  (391ms)
    should pick up existing contracts and NOT deploy new ones into state if MONGO_DB_VERSION is specified (139ms)
  Verify - Monitor
    should prepare the correct data for each contract when verifying on Etherscan (111ms)
    should prepare the correct contract data when pushing to Tenderly Project (405ms)

ZNSAccessController
  Initial Setup
    Should assign GOVERNORs correctly
    Should assign ADMINs correctly
    Should revert when passing 0x0 address to assign roles
  Role Management from the Initial Setup
    GOVERNOR_ROLE should be able to grant GOVERNOR_ROLE
    GOVERNOR_ROLE should be able to revoke GOVERNOR_ROLE
    GOVERNOR_ROLE should be able to grant ADMIN_ROLE
    GOVERNOR_ROLE should be able to revoke ADMIN_ROLE
    ADMIN_ROLE should NOT be able to grant ADMIN_ROLE
    ADMIN_ROLE should NOT be able to revoke ADMIN_ROLE
    ADMIN_ROLE should NOT be able to grant GOVERNOR_ROLE
    ADMIN_ROLE should NOT be able to revoke GOVERNOR_ROLE
    ADMIN_ROLE should be able to grant REGISTRAR_ROLE
    ADMIN_ROLE should be able to grant DOMAIN_TOKEN_ROLE
    ADMIN_ROLE should be able to revoke REGISTRAR_ROLE
    ADMIN_ROLE should be able to revoke DOMAIN_TOKEN_ROLE
    GOVERNOR_ROLE should be able to assign new EXECUTOR_ROLE as admin for REGISTRAR_ROLE
    GOVERNOR_ROLE should be able to assign new EXECUTOR_ROLE as admin for DOMAIN_TOKEN_ROLE
    GOVERNOR_ROLE should be able to make himself a new EXECUTOR_ROLE's admin and assign this role to anyone
    Should revert when setting role admin without GOVERNOR_ROLE
  Role Validator Functions
    #isAdmin() should return true for ADMIN_ROLE
    #isRegistrar() should return true for REGISTRAR_ROLE
    #isDomainToken() should return true for DOMAIN_TOKEN_ROLE
    #isGovernor() should return true for GOVERNOR_ROLE
    #isExecutor() should return true for EXECUTOR_ROLE
    Should revert if account does not have GOVERNOR_ROLE
    Should revert if account does not have ADMIN_ROLE
    Should revert if account does not have REGISTRAR_ROLE
    Should revert if account does not have DOMAIN_TOKEN_ROLE
    Should revert if account does not have EXECUTOR_ROLE

ZNSAddressResolver
   Should NOT let initialize the implementation contract
   Should get the AddressResolver
   Returns 0 when the domain doesnt exist
   Should have registry address correctly set
   Should setRegistry() correctly with ADMIN_ROLE
   Should revert when setRegistry() without ADMIN_ROLE
   Should revert when setAccessController() without ADMIN_ROLE
   Should not allow non-owner address to setAddress
   Should allow owner to setAddress and emit event
   Should allow operator to setAddress and emit event
   Should allow REGISTRAR_ROLE to setAddress and emit event
   Should resolve address correctly
   Should support the IZNSAddressResolver interface ID
   Should support the ERC-165 interface ID
   Should not support other interface IDs
   Should support full discovery flow from zns.registry
```

```
  #setAccessController
     should allow ADMIN to set a valid AccessController
     should allow re-setting the AccessController to another valid contract
     should emit AccessControllerSet event when setting a valid AccessController
     should revert when a non-ADMIN tries to set AccessController
     should revert when setting an AccessController as EOA address
     should revert when setting an AccessController as another non-AC contract address
     should revert when setting a zero address as AccessController
 UUPS
     Allows an authorized user to upgrade the contract
     Fails to upgrade if the caller is not authorized
     Verifies that variable values are not changed in the upgrade process

ZNSCurvePricer
  #encodeConfig and #decodeConfig
     Confirms encoding is the same offchain and onchain
     Confirms decoding is the same offchain and onchain
  #getPrice
     Returns 0 price for a label with no length if label validation is skipped
     Reverts for a label with no length if label validation is not skipped
     Reverts for invalid label if label validation is not skipped
     Returns the max price for domains that are equal to the base length
     Returns the max price for domains that are less than the base length
     Returns expected prices for a domain greater than the base length
     Returns a price even if the domain name is very long
     Returns a price for multiple lengths
     Can Price Names Longer Than 255 Characters
     Can't price a name that has invalid characters
     Returns price as 0 when the baseLength is 0
     Reverts when `precisionMultiplier` in the config is larger than some of the resulting prices
  #validatePriceConfig
     Succeeds when a valid config is provided
     Fails when the curve multiplier is 0 and the base length is 0
     Fails when max length is less than base length
     Fails when maxLength is 0
     Fails when precision muliplier is 0 or greater than 10^18
     Fails when fee percentage is greater than 100%
     Fails when the config bytes are invalid length
  #getRegistrationFee
     Successfully gets the fee for a price
ZNSDomainToken
   should initialize correctly
   should NOT initialize twice
   Should NOT let initialize the implementation contract
  #setRegistry
     Should set ZNSRegistry and fire RegistrySet event
     Should revert if not called by ADMIN
     Should revert if ZNSRegistry is address zero
  #setAccessController
     should allow ADMIN to set a valid AccessController
     should allow re-setting the AccessController to another valid contract
     should emit AccessControllerSet event when setting a valid AccessController
     should revert when a non-ADMIN tries to set AccessController
     should revert when setting an AccessController as EOA address
     should revert when setting an AccessController as another non-AC contract address
     should revert when setting a zero address as AccessController
  External functions
     Should register (mint) the token if caller has REGISTRAR_ROLE
     Should increment the totalSupply when a domain is registered
     Should decrement the totalSupply when a domain is revoked
     Should revert when registering (minting) if caller does not have REGISTRAR_ROLE
     Revokes a token
     #isControlled() should return the correct boolean based on the domain-token ownership config
```

```
  Transfers
    Should update owner for DomainToken and in Registry when transferred normally
    Allows the owner of the domain record in the registry to update the owner
    Fails when non-owner tries to transfer through `safeTransferFrom` and transfers with approval when token and
    ↪  registry record owned by the same address
    Fails when non-owner tries to transfer through `transferFrom`
    #transferFrom() should fail when called by address that only owns the token and not registry record
    #transferOverride() should fail when called by non-registrar
    #transferOverride() should update owner for DomainToken only and not for Registry
    #transferOverride() should revert when transferring to an existing owner
    #transferOverride() should override approvals
    #transferOverride() should emit a `Transfer` event
    #transferOverride() should revert when transferring to address zero (should NOT let to burn the token)
    #transferOverride() should revert when transferring to a contract not implementing #onERC721Received()
    #transferOverride() should revert when transferring to an address that returns incorrect selector from
    ↪  #onERC721Received
  Custom Error Validation
    Only the registrar can call to register a token
    Only authorized can revoke a token
    Should revert when setting access controller if caller does not have ADMIN_ROLE
  Contract Configuration
    Verify token name
    Verify token symbol
    Verify accessController
  Royalties
    should set and correctly retrieve default royalty
    should set and correctly retrieve royalty for a specific token
    #setDefaultRoyalty() should revert if called by anyone other than ADMIN_ROLE
    #setTokenRoyalty() should revert if called by anyone other than ADMIN_ROLE
  Token URIs
    should support individual tokenURIs
    should support baseURI method with tokenURI as 0
    should support baseURI + tokenURI concatenation if both are set correctly
    should return WRONG URI if both baseURI and tokenURI are set as separate links
    should be able to switch from tokenURI to baseURI if tokenURI is deleted
    #setTokenURI() should set tokenURI correctly
    #setTokenURI() should revert if called by anyone other than ADMIN_ROLE
    #setBaseURI() should revert when called by anyone other than ADMIN_ROLE
  ERC-165
    should support IERC721
    should support IERC2981
    should support IERC165
    should not support random interface
  UUPS
    Allows an authorized user to upgrade the contract
    Verifies that variable values are not changed in the upgrade process
    Fails to upgrade if the caller is not authorized

ZNSFixedPricer
  Confirms encoding is the same offchain and onchain
  Confirms decoding is the same offchain and onchain
  #getPrice should return the correct price
  #getPrice() should revert for invalid label when not skipping the label validation
  #getPriceAndFee() should return the correct price and fee (60ms)
  #getFeeForPrice should return the correct fee for a given price
 #validatePriceConfig
    Should pass if price config is valid
    should revert if the price config bytes has invalid length
    should revert if the fee percentage is too high
ZNSRegistry
  Cannot be initialized twice
  Should NOT let initialize the implementation contract
  Should initialize correctly with deployer owning the 0x0 hash domain and should allow to change the ownership later
  Should revert when setting access controller without ADMIN_ROLE
 Audit fix with approved address resolvers
    Adds `address` resolver type and creates a record
    Gets the resolver we currently have assigned to a certain type
    Returns zero for a resolver type that doesn't exist
    Adds a new resolver type
    Deletes a resolver type
```

```
    Operator functionality
       Returns false when an operator is not allowed by an owner
       Returns true when an operator is allowed by an owner
       Returns false when an owner has not specified any operators
       Permits an allowed operator to update a domain record
       Does not permit a disallowed operator to update a domain record
       Does not permit an operator that's never been allowed to modify a record
       #isOperatorFor() should return true for an operator
    Domain records
       Verifies existence of a domain correctly
       Gets a domain record
       Gets a domain owner
       Gets a domain resolver
       Creates a new domain record successfully
       Fails to create a new domain record if the caller does not have REGISTRAR_ROLE
    Setter functions for a domain's record, owner, or resolver
       Cannot update a domain record if the domain doesn't exist
       Can update a domain record if the domain exists
       Cannot update a domain owner if the domain doesn't exist
       Can update a domain owner if the domain exists
       Cannot update a domain resolver if the domain doesn't exist
       Can update a domain resolver if the domain exists
       Cannot update a domain record if the owner is zero address
       Can update a domain record if the resolver is zero address
       cannot update a domain owner if owner is zero address
       Can update a domain resolver if resolver is zero address
       Fails to update a record when caller is not owner or operator
       cannot update a domain's record if not an owner or operator
       cannot update an domain's owner if not an owner or operator
       cannot update a domain's resolver if not an owner or operator
       Can delete record with REGISTRAR_ROLE
       Cannot delete record without REGISTRAR_ROLE
    Event emitters
       Emits an event when an operator is set
       Emits events when a new domain is created
       Emits an event when an existing domain is updated
       Emits an event when a domain's owner is updated
       Emits an event when a domain's resolver is updated
       Emits an event when a domain record is deleted
    #setAccessController
       should allow ADMIN to set a valid AccessController
       should allow re-setting the AccessController to another valid contract
       should emit AccessControllerSet event when setting a valid AccessController
       should revert when a non-ADMIN tries to set AccessController
       should revert when setting an AccessController as EOA address
       should revert when setting an AccessController as another non-AC contract address
       should revert when setting a zero address as AccessController
    UUPS
       Allows an authorized user to upgrade successfully
       Fails when an unauthorized account tries to call to upgrade
       Verifies that variable values are not changed in the upgrade process

  ZNSRootRegistrar
     Gas tests
     Should NOT initialize the implementation contract
     Allows transfer of 0x0 domain ownership after deployment
     Confirms a new 0x0 owner can modify the configs in the treasury and curve pricer
     Confirms a user has funds and allowance for the Registrar
     Should revert when initialize() without ADMIN_ROLE
     Should NOT initialize twice
    General functionality
       #coreRegister() should revert if called by address without REGISTRAR_ROLE
       #setSubRegistrar() should revert if called by address without ADMIN_ROLE
       #setSubRegistrar() should set the correct address
       #setSubRegistrar() should NOT set the address to zero address
       #pauseRegistration() should revert if called by address without ADMIN_ROLE
       #pauseRegistration() should pause the registration and emit #RegistrationPauseSet event
       #pauseRegistration() should NOT pause the registration if already paused
       #unpauseRegistration() should revert if called by address without ADMIN_ROLE
       #unpauseRegistration() should unpause the registration and emit #RegistrationPauseSet
       #unpauseRegistration() should NOT unpause the registration if already unpaused
```

```
    Registers a root domain
       Can NOT register a TLD with an empty name
       Can register a TLD with characters [a-z0-9-]
       Fails for domains that use any invalid character
       Fails when registering during a registration pause when called publicly
       Successfully registers as ADMIN_ROLE during a registration pause
       Successfully registers a domain without a resolver or resolver content and fires a #DomainRegistered event
       Successfully registers a domain with distrConfig and adds it to state properly
       Registers a domain with assigning token owner to a different address
       Stakes and saves the correct amount and token, takes the correct fee and sends fee to Zero Vault
       Takes direct payment when `rootPaymentType` is set to DIRECT
       Stakes the correct amount when `rootPricer` is set to Fixed Pricer and `rootPriceConfig` to fixed price config
       Sets the correct data in Registry
       Fails when the user does not have enough funds
       Disallows creation of a duplicate domain
       Successfully registers a domain without resolver content
       Records the correct domain hash
       Creates and finds the correct tokenId
       Resolves the correct address from the domain
       Should NOT charge any tokens if price and/or stake fee is 0
       Sets the payment config when provided with the domain registration
       Does not set the payment config when the beneficiary is the zero address
    Assigning Domain Token Owners - #assignDomainToken()
       Can assign token to another address and reclaim token if domain hash is owned
       Assigning domain token emits DomainTokenReassigned event
       Cannot assign token if hash is not owned
       Cannot assign token if domain does not exist
       Domain hash can change owner, claim token, transfer, and then be assigned to a diff address again
       Should revert if assigning to existing owner
    Revoking Domains
       Can revoke even if token assigned to a different address
       Charges a protocol fee to the owner as part of the revoke flow
       Revokes without returning funds if domain was registered with DIRECT payment type
       Revokes a Top level Domain, locks distribution and removes mintlist
       Cannot revoke a domain that doesnt exist
       Revoking domain unstakes
       Cannot revoke if Name is owned by another user
       Only token owner can NOT revoke if hash is owned by different address
       After domain has been revoked, an old operator can NOT access Registry
    Bulk Root Domain Registration
       Should register an array of domains using #registerRootDomainBulk (55ms)
       Should revert when register the same domain twice using #registerRootDomainBulk
       Should revert when registering during a registration pause using #registerRootDomainBulk
    State Setters
      #setRegistry
         Should set ZNSRegistry and fire RegistrySet event
         Should revert if not called by ADMIN
         Should revert if ZNSRegistry is address zero
      #setTreasury
         Should set Treasury and fire TreasurySet event
         Should revert if not called by ADMIN
         Should revert if Treasury is address zero
      #setDomainToken
         Should set DomainToken and fire DomainTokenSet event
         Should revert if not called by ADMIN
         Should revert if DomainToken is address zero
      #setRootPricerAndConfig
         should set the rootPricer correctly
         Fails when setting 0x0 address as the new pricer
         Fails when setting an invalid config with a pricer
         Fails when setting an invalid config with a pricer
      #setRootPriceConfig
         should set the rootPricer config correctly
         Fails when setting 0x0 bytes as the new config
         Fails when setting an invalid config with a pricer
         Fails when setting an invalid config with a pricer
      #setAccessController
         should allow ADMIN to set a valid AccessController
         should allow re-setting the AccessController to another valid contract
         should emit AccessControllerSet event when setting a valid AccessController
         should revert when a non-ADMIN tries to set AccessController
         should revert when setting an AccessController as EOA address
         should revert when setting an AccessController as another non-AC contract address
         should revert when setting a zero address as AccessController
```

```
  UUPS
    Allows an authorized user to upgrade the contract
    Fails to upgrade when an unauthorized users calls
    Verifies that variable values are not changed in the upgrade process (39ms)

ZNSStringResolver
  Single state tests
    Should not let initialize the contract twice
    Should correctly attach the string to the domain
    Should setRegistry() using ADMIN_ROLE and emit an event
    Should revert when setRegistry() without ADMIN_ROLE
    Should revert when setAccessController() without ADMIN_ROLE (It cannot rewrite AC address after an incorrect
    ↪  address has been submitted to it)
  New campaign for each test
    Should not allow non-owner address to setString (similar domain and string)
    Should allow OWNER to setString and emit event (similar domain and string)
    Should allow OPERATOR to setString and emit event  (different domain and string)
    Should support the IZNSAddressResolver interface ID
    Should support the ERC-165 interface ID
    Should not support other interface IDs
   #setAccessController
      should allow ADMIN to set a valid AccessController
      should allow re-setting the AccessController to another valid contract
      should emit AccessControllerSet event when setting a valid AccessController
      should revert when a non-ADMIN tries to set AccessController
      should revert when setting an AccessController as EOA address
      should revert when setting an AccessController as another non-AC contract address
      should revert when setting a zero address as AccessController
    UUPS
      Allows an authorized user to upgrade the StringResolver
      Fails to upgrade if the caller is not authorized
      Verifies that variable values are not changed in the upgrade process

ZNSSubRegistrar
  Single Subdomain Registration
    Sets the payment config when given
    Does not set the payment config when the beneficiary is the zero address
    should revert when trying to register a subdomain before parent has set it's config with FixedPricer
    should revert when trying to register a subdomain before parent has set it's config with CurvePricer
    should revert when registering during a registration pause when called publicly
    should register successfully as ADMIN_ROLE during a registration pause
    should register subdomain with the correct tokenURI assigned to the domain token minted (69ms)
    Can register a subdomain with characters [a-z0-9-]
    should register a subdomain with token assigned to a different address if provided
    Fails for a subdomain that uses any invalid characters
    should revert when trying to register a subdomain under a non-existent parent
    should register subdomain with a single char label
    should register subdomain with a label length of 100000 chars [ @skip-on-coverage ] (3690ms)
    should revert when user has insufficient funds
    should revert when user has insufficient allowance
    should revert on payment when parent's beneficiary has not yet been set and when stakeFee is > 0 (1119ms)
  Bulk Subdomain Registration
    Should #registerSubdomainBulk and the event must be triggered (79ms)
    Should register multiple NESTED subdomains using #registerSubdomainBulk (190ms)
    Should revert when register the same domain twice using #registerSubdomainBulk
    Should revert with 'ZeroAddressPassed' error when 1st subdomain in the array has zerohash
    Should register a mix of nested and non-nested subdomains and validates hashes (101ms)
    Should revert when registering during a registration pause using #registerSubdomainBulk
  Operations with domain paths
    should register a path of 6 domains with different configs (60ms)
    should be able to register multiple domains under multiple levels for the same owner (189ms)
    should revoke lvl 6 domain without refund, lock registration and remove mintlist
    should revoke lvl 5 domain with refund
    should register a new 2 lvl path at lvl 3 of the existing path (76ms)
    should revoke lvl 3 domain (child) with refund after lvl 2 (parent) has been revoked
    should let anyone register a previously revoked domain (68ms)
    should NOT register a child (subdomain) under a parent (root domain) that has been revoked
    should NOT register a child (subdomain) under a parent (subdomain) that has been revoked
    should allow setting a new config and start distributing subdomain when registering a previously revoked parent
    ↪  (66ms)
```

```
  Token movements with different distr setups
    FixedPricer - StakePayment - stake fee - 5 decimals (43ms)
    Does not charge the owner of a parent domain when they revoke a subdomain
    FixedPricer - StakePayment - no fee - 18 decimals (47ms)
    FixedPricer - DirectPayment - no fee - 8 decimals (48ms)
    CurvePricer - StakePayment - stake fee - 13 decimals (47ms)
    CurvePricer - StakePayment - no fee - 2 decimals (54ms)
    CurvePricer - DirectPayment - no fee - 18 decimals (44ms)
    FixedPricer + DirectPayment with price = 0 - should NOT perform any transfers (46ms)
    CurvePricer + DirectPayment with price = 0 - should NOT perform any transfers (44ms)
    CurvePricer + StakePayment with price = 0 - should NOT perform any transfers (52ms)
    FixedPricer + StakePayment with price = 0 - should NOT perform any transfers (43ms)
    Setting price config in incorrect decimals triggers incorrect pricing (65ms)
  Registration access
    should allow parent owner to register a subdomain under himself even if accessType is LOCKED
    should allow parent owner's operator to register but assign subdomain hash to the owner
    should NOT allow others to register a domain when parent's accessType is LOCKED
    should allow anyone to register a domain when parent's accessType is OPEN
    should ONLY allow mintlisted addresses and NOT allow other ones to register a domain when parent's accessType is
    ↪  MINTLIST (49ms)
    #updateMintlistForDomain() should NOT allow setting if called by non-authorized account or registrar
    #updateMintlistForDomain() should fire a #MintlistUpdated event with correct params
    should switch accessType for existing parent domain
    should NOT allow to register subdomains under the parent that hasn't set up his distribution config
  Existing subdomain ops
    should NOT allow to register an existing subdomain that has not been revoked
    should NOT allow revoking when the caller is NOT an owner of hash in Registry
    should allow to UPDATE domain data for subdomain
    should TRANSFER ownership of a subdomain and let the receiver revoke with REFUND
   #setDistributionConfigForDomain()
      should re-set distribution config for an existing subdomain
      should NOT allow to set distribution config for a non-authorized account
      should revert if pricerContract is passed as 0x0 address
   #setPricerDataForDomain()
      should re-set pricer contract for an existing subdomain
      should NOT allow setting for non-authorized account
      should NOT set pricerContract to 0x0 address
   #setPaymentTypeForDomain()
      should re-set payment type for an existing subdomain
      should NOT allow setting for non-authorized account
      should emit #PaymentTypeSet event with correct params
  State setters
    Should NOT let initialize the implementation contract
    #setRootRegistrar() should set the new root registrar correctly and emit #RootRegistrarSet event
    #setRootRegistrar() should NOT be callable by anyone other than ADMIN_ROLE
    #setRootRegistrar should NOT set registrar as 0x0 address
    #setRegistry() should set the new registry correctly and emit #RegistrySet event
    #setRegistry() should not be callable by anyone other than ADMIN_ROLE
    #pauseRegistration() should pause the registration process and emit #RegistrationPauseSet event
    #pauseRegistration() should not be callable by anyone other than ADMIN_ROLE
    #pauseRegistration() should not allow to pause if already paused
    #unpauseRegistration() should unpause the registration process and emit #RegistrationPauseSet event
    #unpauseRegistration() should not be callable by anyone other than ADMIN_ROLE
    #unpauseRegistration() should not allow to unpause if already unpaused
   #setAccessController
      should allow ADMIN to set a valid AccessController
      should allow re-setting the AccessController to another valid contract
      should emit AccessControllerSet event when setting a valid AccessController
      should revert when a non-ADMIN tries to set AccessController
      should revert when setting an AccessController as EOA address
      should revert when setting an AccessController as another non-AC contract address
      should revert when setting a zero address as AccessController
  UUPS
    Allows an authorized user to upgrade the contract
    Fails to upgrade if the caller is not authorized
    Verifies that variable values are not changed in the upgrade process (46ms)
    Allows to add more fields to the existing struct in a mapping
```

```
  ZNSTreasury
     Should initialize correctly
     should NOT initialize twice
     Should NOT let initialize the implementation contract
     should NOT deploy/initialize with 0x0 addresses as args (56ms)
    #stakeForDomain()
       Stakes the correct amount
       Should revert if called from an address without REGISTRAR_ROLE
       Should fire StakeDeposited event with correct params
    #unstakeForDomain()
       Unstakes the correct amount and saves the correct token
       Should revert if called from an address without REGISTRAR_ROLE
    #processDirectPayment()
       should process payment correctly with paymentConfig set
       should revert if paymentConfig not set
       should revert if called by anyone other than REGISTRAR_ROLE
       should emit DirectPaymentProcessed event with correct params
    #setPaymentConfig(), BeneficiarySet and PaymentTokenSet
       should set payment config for an existing subdomain
       should NOT allow setting for non-authorized account
       should NOT set token or beneficiary to 0x0 address
    #setBeneficiary() and BeneficiarySet event
       Should set the correct address of Zero Vault
       Should revert when called by anyone other than owner or operator
       Should revert when beneficiary is address 0
    #setPaymentToken() and PaymentTokenSet event
       Should set the correct address
       Should revert when called by anyone other than owner or operator
       Should revert when paymentToken is address 0
    #setAccessController
       should allow ADMIN to set a valid AccessController
       should allow re-setting the AccessController to another valid contract
       should emit AccessControllerSet event when setting a valid AccessController
       should revert when a non-ADMIN tries to set AccessController
       should revert when setting an AccessController as EOA address
       should revert when setting an AccessController as another non-AC contract address
       should revert when setting a zero address as AccessController
    #setRegistry() and RegistrySet event
       Should set the correct address of Registry
       Should revert when called from any address without ADMIN_ROLE
       Should revert when registry is address 0
    UUPS
       Allows an authorized user can upgrade the contract
       Fails when an unauthorized user tries to upgrade the contract
       Verifies that variable values are not changed in the upgrade process

  Transaction Gas Costs Test

      Root Domain Price:
        Gas Used: 540794
        Gas Diff: 0

     Root Domain Price

      Subdomain Price:
        Gas Used: 544638
        Gas Diff: 37

     Subdomain Price


  476 passing (1m)

$ yarn mongo:stop
warning ../../../../../package.json: No license field
$ docker-compose stop
Stopping zns_mongo_1 ... done
  Done in 96.70s.
```

# 9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

**Blockchain Security:** At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

**Blockchain Core Development:** Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

**DevOps and Infrastructure Management:** Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

**Cryptography Research:** At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

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

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

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

**Learn more about us at nethermind.io**.

### General Advisory to Clients

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

### Disclaimer

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.