Quantstamp

Zero Name Service (ZNS)

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	Address Resolver
Timeline	2023-10-12 through 2023-10-20
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	architecture.md flows.md roles.md
Source Code	• zer0-os/zNS #46f632e
Auditors	• Jennifer Wu Auditing Engineer • Hytham Farah Auditing Engineer • Mustafa Hasan Senior Auditing Engineer

Documentation quality	High
Test quality	High
Total Findings	18 Fixed: 10 Acknowledged: 7 Mitigated: 1
High severity findings	1 Fixed: 1
Medium severity findings	6 Fixed: 3 Acknowledged: 3
Low severity findings	5 Fixed: 3 Acknowledged: 1 Mitigated: 1
Undetermined severity findings	1 Fixed: 1
Informational findings	5 Fixed: 2 Acknowledged: 3

Summary of Findings

This report outlines the audit of the Zero Name Service (ZNS) protocol by Quantstamp. Zero Name Service offers a decentralized naming system on the Ethereum mainnet that grants human-readable names to entities. ZNS allows users to link their blockchain wallets, smart contracts, or on-chain data to their chosen name. The ZNS protocol allows users to register domains using either staking tokens or direct payment, as determined by the domain owner. The domain's pricing is set by a pricer contract designated by the domain owner. Furthermore, the domain owner can also specify an address resolver to correctly map the subdomain to its associated address.

The audit identified a total of 18 findings. The audit highlighted issues such as domain impersonation because of inadequate label validation (ZNS-1) and domain spoofing due to the absence of on-chain label normalization (ZNS-2). Furthermore, the protocol risks fund insolvency due to potential token incompatibility (ZNS-3), and flaws in pricing calculations could lead to unexpected domain costs for users (ZNS-4).

We found the test coverage to be extensive and high quality, but we recommend adding a test case for each high and medium issue identified in this report to verify the fixes. Additionally, we advise the client to update all documentation to align with the latest ZNS implementation and to address or consider all the findings provided in this report.

Fix Review: The client addressed all issues from ZNS-1 to ZNS-18, implementing necessary fixes or providing detailed explanations for acknowledgments. For domain impersonation and spoofing issues (ZNS-1 and ZNS-2), strict label validation was added, limiting characters to a-z, 0-9, and hyphens. The protocol risk of fund insolvency due to potential token incompatibility (ZNS-3) was acknowledged, with explanations and updated user documentation highlighting the associated risks. Additionally, the flaws in pricing calculations (ZNS-4) were corrected. The client also provided test cases for each fix to ensure the effectiveness of the fixes.

ID	DESCRIPTION	SEVERITY	STATUS
ZNS-1	**Potential Domain and Subdomain Impersonation Due to Absence of String Validation**	● High	Fixed
ZNS-2	**Potential for Domain Spoofing Due to Lack of on-Chain Name Normalization**	● Medium	Fixed
ZNS-3	**Risk of Insolvency Due to Inaccurate Accounting with**	● Medium	Acknowledged

ID	DESCRIPTION	SEVERITY	STATUS
	Deflationary or Rebasing Tokens		
ZNS-4	Potential to Undercharge for Domain Label with Lengths of `maxLength`	● Medium	Fixed
ZNS-5	Precision Loss Can Result in Lower than Expected Purchase Price	● Medium	Acknowledged
ZNS-6	Get a Free Domain with Undefined Price Configuration	● Medium	Fixed
ZNS-7	Domain Pricer Contract Can Charge More than Expected when Purchasing Domain	● Medium	Acknowledged
ZNS-8	Unintended Mintlist After Domain Revocation	● Low	Fixed
ZNS-9	Funds Can Be Sent to Zero Address	● Low	Fixed
ZNS-10	Previous Owner Can Update Domain Configuration	● Low	Acknowledged
ZNS-11	Uninitialized Implementation Contract	● Low	Fixed
ZNS-12	Missing Input Validation	● Low	Mitigated
ZNS-13	Deviation From Traditional Domain and Subdomain Representation	● Informational	Acknowledged
ZNS-14	Potential for Disrupted Access Control Due to Registry Mismatch	● Informational	Acknowledged
ZNS-15	Privileged Roles	● Informational	Fixed
ZNS-16	Upgradability	● Informational	Acknowledged
ZNS-17	Unlocked Pragma	● Informational	Fixed
ZNS-18	Domain Address Resolver Can Be Malicious	● Undetermined	Fixed

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Disclaimer
Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification

- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
    1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
    1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Findings

ZNS-1
Potential Domain and Subdomain Impersonation Due to Absence of String Validation

● High ⓘ Fixed

Update
The client fixed the issue in `5b97484533cdfee1aca8ebf90090357e77d9e5bd` by using `StringUtils.validate()`, a newly implemented function. This function ensures that strings in the `registerRootDomain()` and `registerSubdomain()` functions strictly contain only lowercase letters (a-z), numerals (0-9), or hyphens.

File(s) affected: `ZNSRootRegistrar.sol`, `ZNSSubRegistrar.sol`

Description: The `ZNSRootRegistrar.registerRootDomain()` function does not validate if the provided name contains a dot. Consequently, if a user has registered a domain like `0://wilder`, another user could register `0://wilder.fundme`, which might mislead users into believing it is a legitimate subdomain of the original domain. This impersonation risk arises because the genuine subdomain registration logic resides solely in the `ZNSSubRegistrar.sol` contract.

Furthermore, this oversight in validation extends to the `ZNSSubRegistrar.registerSubdomain()` function. An attacker can register multi-level subdomains all at once. As an example, given the registered domain `0://wilder`, an attacker could directly register a third-level subdomain like `0://wilder.2ld.3ld.4ld` without sequentially registering `0://wilder.2ld` and `0://wilder.2ld.3ld`.

Recommendation: Ensure the string validation process verifies that the input domain or subdomain name does not contain any dots. This would prevent users from bypassing the intended hierarchical registration process and deter potential domain or subdomain impersonation.

ZNS-2
Potential for Domain Spoofing Due to Lack of on-Chain Name Normalization

● Medium ⓘ Fixed

Update
The client fixed the issue in `5b97484533cdfee1aca8ebf90090357e77d9e5bd` by using `StringUtils.validate()`, a newly implemented function. This function ensures that strings in the `registerRootDomain()` and `registerSubdomain()` functions strictly contain only lowercase letters (a-z), numerals (0-9), or hyphens.

File(s) affected: `ZNSRootRegistrar.sol`, `ZNSSubRegistrar.sol`

Description: The `ZNSRootRegistrar.registerRootDomain()` function and `ZNSSubRegistrar.registerSubdomain()` subdomain registration function does not account for domain name string variations, especially concerning character casing. This lack of validation means that domain names, which appear identical after normalization, are recognized as distinct entities on-chain due to their differing hash representations.

For instance, both `wilder` and `wIlder` can be registered as separate domains. However, when client-side normalization interprets both names as `wilder`, both domains could resolve to the same address, creating potential impersonation risks.

Recommendation: Ideally, an on-chain name normalization within the contracts should be implemented. Convert all domain name inputs to lowercase before determining their `domainHash` and completing the domain registration. This would ensure that domains with varying letter cases but identical letters will generate the same `domainHash`, precluding duplicate registrations.

However, considering the potential gas implications of on-chain string operations and their resulting cost, the following are some mitigation measures that the client can implement to protect users:

1. Draft comprehensive documentation emphasizing the potential issues arising from domain name casing and the effects of client-side normalization.
2. Provide warnings or guidelines during the domain registration phase to inform users of the implications of registering domain names with diverse casings.
3. Introduce client-side measures that can detect potential registration conflicts and offer suitable warnings or recommendations.

ZNS-3
Risk of Insolvency Due to Inaccurate Accounting with Deflationary or Rebasing Tokens

● Medium Acknowledged

Update

The client acknowledged the issue and provided the following explanation:

> Outline possible problems in the docs. Answer: We consider this a special rare case when a parent domain owner uses rebasing or deflationary token as payment. In order for a subdomain candidate to pay in this token, he first has to purchase it, making him more knowledgeable of the token he is about to use. Also, this is tied to a specific community which operates on a specific token, so participants should be aware of the nature of the token. To solve this we would need to centralize the system more with a token whitelist, which, we believe, goes against our goals of leaning more towards decentralization. This would also add complexity to the system and take the freedom of building your community on your own rules away. We will add and expand the below paragraph in our user-facing docs.

The client plans to provide the following documentation to users:

> Docs: When purchasing a subdomain, be careful and review all the rules set by the parent domain owner, under which you purchase your domain. zNS system is allowing a freedom for a parent domain owner to choose whichever token they like to be used for payments. Please note, that in case of staking, the amount you paid is fixed and saved in the ZNSTreasury contract's state and if you decide to revoke your subdomain later, this is the exact amount you will get back in the same token you staked in. In case a parent domain owner used rebasing or deflationary token for stake payments, the amount you will withdraw upon revocation may not hold the same value it originally did based on the token's total supply at any given time. We recommend to thoroughly research the community you are entering with subdomain purchase and especially the token and the economy this community uses. zNS provides the way to freely set your own rules and tokens for any domain owner, but can not guarantee how these tokens and economies will work outside of zNS.

File(s) affected: `ZNSTreasury.sol`

Description: The `ZNSTreasury.stakeForDomain()` function accounts for staked amounts based on the tokens deposited. If deflationary tokens, which automatically subtract fees on transfer, or rebasing tokens, which can adjust balances algorithmically, are used, this can lead to discrepancies between the recorded `stakedForDomain.stakedAmount` and the actual reserve. Furthermore, since the protocol accepts any token, a malicious token could exacerbate this discrepancy. These inaccuracies can lead to insolvency issues during unstaking.

Recommendation: To address this risk, either limit the types of currencies accepted when registering a subdomain to exclude deflationary or rebasing tokens, or adjust the protocol's accounting mechanisms to properly handle deflationary or rebasing tokens.

ZNS-4
Potential to Undercharge for Domain Label with Lengths of `maxLength`

● Medium Fixed

Update

The client fixed the issue in `14749feacdd9304028c965182adf59ed7d570c8d` by adjusting `_validateConfig()` to verify prices at `config.maxLength` instead of `config.maxLength - 1`.

File(s) affected: `ZNSCurvePricer.sol`

Description: The `ZNSCurvePricer._getPrice()` function is designed to determine the price of domain labels based on their lengths, following an asymptotic decline in price as length increases. However, there is a flaw in the function `_validateConfig`. While the function is intended to return a price no less than the `minPrice`, an edge case exists when the domain length is exactly `maxLength`. This issue emerges because the validation function `_validateConfig` only checks the price at `maxLength - 1`, assuming the price at `maxLength` would always be above `minPrice`. Therefore, the returned price might be less than the specified `minPrice` and the protocol might unintentionally charge users less than the intended `minPrice` when they acquire domains of exact length `maxLength`. This can lead to potential revenue loss for the domain owner.

Exploit Scenario: Consider the following pricing configuration:

- `baseLength` : 5
- `maxLength` : 10
- `maxPrice` : 10
- `minPrice` : 5.5

Consider the prices obtained for the following domain label lengths:

- Price for a domain of length 9 (`maxLength - 1`): `5 * 10 / 9 = 5.56` (rounded), which is greater than 5.5
- Price for a domain of length 10 (`maxLength`): `5 * 10 / 10 = 5` , which is below the `minPrice` of 5.5.

The pricing configuration passes `_validateConfig` because the price calculated for length = 9 (`maxLength - 1)` is greater than `minPrice` of 5.5.

Recommendation: Modify the `_getPrice()` function to return `config.minPrice` for domain lengths equal to or exceeding config.maxLength. Alternatively, adjust `_validateConfig()` to verify prices at `config.maxLength` instead of `config.maxLength - 1` . Another approach could be to introduce a validation mechanism that enforces the condition `minPrice <= (baseLength * maxPrice) / maxLength` whenever `baseLength` , `maxPrice` , or `maxLength` are updated.

ZNS-5

Precision Loss Can Result in Lower than Expected Purchase Price

● Medium ⓘ Acknowledged

Update

The client acknowledged the precision loss issue in pricer configurations and will implement front-end measures to prevent user misconfigurations. They will also enhance documentation and warnings about risks associated with direct contract interactions. Additionally, the client clarified that the design involving division before multiplication is an intentional choice for truncation purposes.

The client provided the following explanation:

> priceConfig.precisionMultiplier is an optional parameter in ZNSCurvePricer that can be set by the domain owner/operator to truncate the unnecessary decimals off the price. If precision for calculation is not needed for a domain owner, precisionMultiplier should be set as 1, that will avoid truncation altogether.
>
> If truncation is needed precisionMultiplier should be calculated as: 10 ^ (tokenDecimals - requiredPrecision). e.g. if the token has 8 decimals and you want to only leave 2 decimals, the precisionMultiplier would be 10 ^ (8 - 2) = 10 ^ 6. This would result in a price value of 1.23456789 TOKEN to be returned as 1.23 TOKEN.
>
> Please not that precisionMultiplier is NOT a direct value of precision you want, but a result of subtraction of precision value from the value of decimals of the token chosen as a power of 10! If truncation is not needed, precisionMultiplier should be set to 1. It will be set to 1 as a default value for all newly minted domains, UNLESS the domain registrant specifically sets it to the value they choose. The precisionMultiplier can NOT be set to 0, because it would produce a price of 0 disregarding the rest of the config! The precisionMultiplier can NOT be set to a value greater than 10 ^ 18! The precisionMultiplier can, but should NOT be set to a value higher than 10 ^ decimals of the token used for payments and pricing. Doing this would result in an incorrect price calculation followed by the incorrect payment. Please note that the correct return from getPrice() will be dependent on the domain owner/operator to set the precisionMultiplier correctly or leave it as a default 1.

File(s) affected: `ZNSCurvePricer.sol`

Description: The primary concern identified in the `ZNSCurvePricer._getPrice()` function is the potential incompatibility between the `precisionMultiplier` and the token's decimals, leading to incorrect pricing calculations. This incompatibility can result in a significant loss of precision, especially evident when using a high `precisionMultiplier` (e.g., 10^18) with tokens that have a lower decimal precision (e.g., USDC with 6 decimal places). This precision loss can result in the computed price being truncated to zero, as demonstrated in the given exploit scenario.

Exploit Scenario: Consider the following pricing configuration:

- `baseLength` : 36
- `length` : 911
- `maxLength` : 1000
- `maxPrice` : 983678523 `USDC` (6 decimal places)
- `minPrice` : 0
- `precisionMultiplier` : 1000000000000000000 (10^18)

The intermediate price computation `(baseLength * maxPrice / length)` computes to `38872038`. However, the subsequent division by the high `precisionMultiplier` of 10^18 truncates this value to zero. This results in the domain's final computed price becoming `0` USDC, whereas the correct expected price, without the precision loss, should have been `38872038` USDC.

Recommendation: To mitigate this issue, it is recommended to implement a validation mechanism that checks the compatibility between the `precisionMultiplier` and the token decimal places. This validation should be incorporated within the _validateConfig() function or a similar configuration validation process. Ensuring this compatibility will prevent configurations that could lead to significant precision loss.

ZNS-6 Get a Free Domain with Undefined Price Configuration

• Medium ⓘ Fixed

Update

The client fixed the issue in `f42f8ded71b9e28406735b0a77081cc856bd4cf8` by adding an `isSet` boolean variable to the `PriceConfig` struct. This boolean is validated in `getPrice()` function before returning the price from the `ZNSFixedPricer` and `ZNSCurvePricer` contracts. This boolean variable can be only set in the function `setPriceConfig()`.

File(s) affected: `ZNSCurvePricer.sol`, `ZNSFixedPricer.sol`

Description: The `getPrice()` function within the ZNSCurvePricer contract calculates the domain's price based on the `CurvePriceConfig`. If no pricing configuration is set for a domain, the system may inadvertently return a price of zero due to the default behavior of Solidity which initializes struct values to zero. When a user attempts to register a subdomain, the `ZNSSubRegistrar.registerSubdomain()` function, present in an external contract, invokes the `getPrice()` function. If the domain pricing configuration is missing or not correctly defined, the domain may be registered for free.

Recommendation: Introduce a mechanism to check if a domain's pricing configuration has been explicitly defined before calculating its price. Consider adding a boolean variable, e.g., `isConfigSet`, within the `CurvePriceConfig` struct to indicate if the pricing configuration for a domain has been intentionally set.

ZNS-7 Domain Pricer Contract Can Charge More than Expected when Purchasing Domain

• Medium ⓘ Acknowledged

Update

The client acknowledged the issue and provided the following explanation:

> We believe this problem can easily be avoided with proper and responsible approvals from the users, which has to be communicated in the documentation. We provide a view function getPrice() which any user can utilize to figure out the correct approval amount for his future domain before he registers. Zero frontend application would also get this value from a contract before building the approval tx for the user (if needed) and presenting him with values. In the case of the correct approval, any frontrunned price that is higher than the approval amount would trigger a revert on the token transfer.

The client plans to provide the following documentation to users:

> Docs: Before purchasing your subdomain, please ensure you are fully aware of it's price and all fees by calling the respective Pricer contract's getPriceAndFee() function and ZNSCurvePricer.getFeeForPrice() to get the amount for protocolFee. In order to pay for the subdomain you have to approve this price along with all the fees for ZNSTreasury contract that will perform token transfers. Please note, that you should ALWAYS approve only the exact amount you have reviewed and agree with. Do NOT approve ZNSTreasury to spend more tokens that are required by the domain price + fees, since it can open you to a possible problem of parent owner trying to charge you over the price you've reviewed. Approval of the exact amounts is a general rule while dealing with smart contracts or other wallets and it protects you from malicious charges and spending more than you planned.

File(s) affected: `ZNSSubRegistrar.sol`

Description: The `ZNSSubRegistrar` contract provides a setter function, `setPricerContractForDomain()`, allowing the domain owner/operator to change the pricer contract for a specific domain. With this ability, a malicious domain owner could potentially front-run a domain purchase transaction, swapping out the pricer contract for one that charges an unexpectedly high price. This is problematic if a user purchasing the domain approves a transaction for a significantly higher amount than required or unlimited approval, as they may end up being charged much more than anticipated.

Recommendation: It is essential to clearly communicate this risk to all users of the protocol, advising them to be cautious when setting approval amounts and to be aware of the potential for unexpected price hikes. We recommend some mitigation measures that the client can consider to protect users (but not limited to):

1. Introducing a time delay after modifying the pricer contract for a domain. This offers a window for potential buyers to observe and react to any price changes.
2. Allowing users to specify a maximum acceptable purchase price. If the price set by the pricer contract surpasses this user-defined threshold, the transaction could be automatically rejected.

ZNS-8 Unintended Mintlist After Domain Revocation

• Low ⓘ Fixed

Update

The client fixed the issue in `8f8bdffc5bd2b2415d2fd6246e0946acfc9468d2` by implementing a `Mintlist` struct, which includes mapping of address approvals `list` and an `ownerIndex` index. This index is incremented each time the `clearMintListForDomain()` function is called, which is also invoked within the `revokeDomain()` function. Through this mechanism, each change in domain ownership will be a new mapping in the Mintlist, based on their respective `ownerIndex`. We also recommend including `ownerIndex` in the event `MintlistUpdated`, so it is clear which owner updated the candidate when using the function `updateMintlistForDomain()`.

File(s) affected: `ZNSRootRegistrar.sol`, `ZNSSubRegistrar.sol`

Description: When a domain is revoked via `ZNSRootRegistrar.revokeDomain()`, several actions occur: the access type is set to `AccessType.LOCKED`, its token is burnt, its registry record is removed, and staked funds, if any, are returned to the domain beneficiary. However, if a new owner purchases this domain and sets the `AccessType` to `MINTLIST`, the protocol will default to the previous owner's mintlist configuration, which is not cleared during revocation. The new domain owner, unaware, can be constrained by the previous owner's access settings until they use `updateMintlistForDomain()` to clear the previous allowance.

Recommendation: To address the issue, implement a mechanism in the `ZNSRootRegistrar.revokeDomain()` function to clear the mintlist associated with the revoked domain. This ensures that subsequent domain owners start with a clean slate, preventing unintentional access constraints inherited from previous configurations.

ZNS-9 Funds Can Be Sent to Zero Address

• Low ⓘ Fixed

Update

The client fixed the issue in `90c745b90c80dae0de74da4a9d2fd6814d18b2fc` by checking the `beneficiary` address before the transfer.

File(s) affected: `ZNSTreasury.sol`

Description: The protocol permits domain owners to designate a beneficiary address for payments associated with a specific domain, identified by its `domainHash` in the `ZNSTreasury` through functions `setPaymentConfig()` and `setBeneficiary()`. However, if a beneficiary is not set for a `domainHash`, funds sent to purchase a subdomain under this parent `domainHash` will be directed to a zero address. While OpenZeppelin's ERC20 implementation includes checks against transfers to a zero address, the treasury is designed to accept any ERC20 tokens, which might not have such safeguards.

Recommendation: Implement beneficiary validation prior to fund transfer to ensure beneficiary addresses are set, regardless of the ERC20 token's own safeguards, to prevent potential loss of funds.

ZNS-10 Previous Owner Can Update Domain Configuration

• Low ⓘ Acknowledged

Update

From discussions with the client, this design is intentional and is intended to allow an owner the flexibility to distribute their ownership. The client acknowledged the issue and provided the following explanation:

> Please note that when you are buying a zNS domain on any marketplace or inheriting it from a previous owner, you are inheriting all distribution rules, price and payment configs and subdomains of that domain. While parent domain owners in zNS have no power over existing children, you ARE able to reset all existing distribution configs and Resolver records to your own, which you should absolutely do if you do not agree with previous ones set by the previous owner. But in order to do that you HAVE to first reclaim a full ownership of your new domain, by calling ZNSRootRegistrar.reclaim(). In order to call this function you have to already be the owner of the token you received as a representation of the domain transfer. Upon calling this function (or using Zero UI) you will become the full owner of the Name and the Token of this domain and will be able to reset any configs and data records related to it. Keep in mind that before you reclaim your full ownership, the previous owner of the Name (ZNSRegistry) is still able to set their own config values for distribution or name resolution, but they can easily be overwritten by you after you reclaim. You will start getting paid for you child subdomain purchases ONLY AFTER you reclaimed, and if previous configs have not been set, setting your own configs.

File(s) affected: `ZNSRegistry.sol`, `ZNSRootRegistrar.sol`

Description: The ownership transfer process of a `domainHash` in the protocol requires two distinct actions: transferring the corresponding ERC721 token and calling the `ZNSRootRegistrar.reclaimDomain()` function. This approach introduces a potential oversight issue. If an owner transfers the ERC721 token but the new owner fails to immediately invoke `reclaimDomain()`, there exists a window of time where the original owner or operator, despite not having the ERC721 token, retains the ability to make impactful changes via the `ZNSSubRegistrar` contract. During this window, the previous owner or operator could, for instance, update configuration settings through methods such as `setDistributionConfigForDomain()` and `setPricerContractForDomain()`.

Recommendation: Reassess the ownership design to ensure it inherently mitigates the risks of unintended split ownership. If maintaining the current design, incorporate fail-safes or warnings prompting the new owner to complete the `reclaimDomain()` step immediately after receiving the ERC721 token. Furthermore, consider implementing robust access control checks to validate domain ownership consistency across both the ERC721 token and ownership before any configuration changes can be made.

ZNS-11 Uninitialized Implementation Contract

• Low ⓘ Fixed

> **Update**
> The client fixed the issue in `482467968891af42d8dc6798cff6a2d29e308931`.

File(s) affected: `All upgradeable contracts`

Description: The implementation contracts may not consistently ensure their initializer is disabled. This vulnerability allows malicious actors to call the `initialize()` function on the implementation contracts. Though the `initialize()` function is guarded by the "initializer" modifier to ensure it is invoked only once, this protection is valid only from the proxy contract's storage layout. The logic contract remains uninitialized from its direct perspective because `initialize()` is not directly activated within it. Consequently, unauthorized entities can call the `initialize()` function and alter state variables of the implementation contract.

Recommendation: Although the UUPSUpgradeable self-destruct vulnerability has been mitigated in OpenZeppelin's contracts in v4.3.2, we recommend adding `_disableInitializers()` to the constructor of all upgradeable contracts.

ZNS-12 Missing Input Validation

• Low ⓘ Mitigated

> **Update**
> The client added most of the validations in `e2c258bcbfdee1f4effb54f5a90214ac4c17a101` and provided the following explanation:
>
> > 3 It does not necessarily need to be a contract. An EOA will probably be used to run the deploy pipeline, then the ownership should be switched to a multisig. I fixed the NatSpec.
> >
> > 4 It is expected to introduce multiple different resolvers each of which will have it's own type and methods. e.g. AddressResolver contract has getAddress() , but a StringResolver might have getString() method. We can't predict which types of Resolvers will release at which time, so creating an overall Resolver type that will always work for this validation doesn't really seem possible or may not be complete or may require upgrade of the contracts not related to Resolvers if any changes are needed.
> >
> > 5 The worst-case scenario is if a required function is missing, then the function will revert, so the validation is not necessary.

File(s) affected: `ZNSAccessController.sol`, `ZNSDomainToken.sol`, `ZNSRegistry.sol`, `ARegistryWired.sol`, `ZNSCurvePricer.sol`, `ZNSFixedPricer.sol`

Description: It is crucial to validate inputs, even if the inputs come from trusted addresses, to avoid human error. A lack of robust input validation can only increase the likelihood and impact in the event of mistakes.

Following is the list of places that can potentially benefit from stricter input validation:
1. Fixed `ZNSAccessController.sol#77` : the `addresses` of the `_grantRoleToMany()` should not be zero.
2. Fixed `ZNSDomainToken.sol#122` : the `royaltyFraction` of the `setDefaultRoyalty()` should be equal to or less than base `10_000`.
3. Fixed `ZNSRegistry.sol#60` : the `msg.sender` of the `initialize()` should be a contract. The NatSpec claims the owner of 0x0 is a multisig.
4. Acknowledged `ZNSRegistry.sol#262` : the `resolver` interface of the `_setDomainResolver` should be validated before setting the resolver address.
5. Acknowledged `ARegistryWired.sol#35` : the `registry_` interface of the `_setRegistry()` should be validated before setting the registry.
6. Fixed `ZNSCurvePricer.sol#109` : the `priceConfig.feePercentage` of the `setPriceConfig()` should be equal to or less than `PERCENT_BASIS`.
7. Fixed `ZNSCurvePricer.sol#246` : the `feePercentage` of the `setFeePercentage()` should be equal to or less than `PERCENT_BASIS`.
8. Fixed `ZNSFixedPricer.sol#130` : the `feePercentage` of the `_setFeePercentage()` should be equal to or less than `PERCENT_BASIS`.

Recommendation: Add the validations and checks listed in the description.

ZNS-13
Deviation From Traditional Domain and Subdomain Representation

● Informational ⓘ Acknowledged

Update

The client acknowledged the issue and provided the following explanation:

> This is a very intentional feature of the protocol. ZNS is built to function beyond the internet domain archetype, as a routing protocol, indexer, community management tool, and more. As such, it hews to longstanding file system conventions of left-to-right, top-down organization where the root is located on the left and the latest branch on the right. We anticipate ZNS accommodating many, many nested iterations of World domains, parent domains, and subdomains — where every subdomain can itself become a parent domain — and the Web2/Web3 right-to-left domain convention does not lend itself well to iterative extensibility. In this sense, even though we refer to ZNS tokens as 'domains,' it is better to think of the ZNS protocol as a distributed global index or internet file system.

The client plans to provide the following documentation to users:

> Docs: Please note that ZNS denominates domain name hierarchies in a left-to-right, top-down fashion, i.e., for the domain 0://hello.goodbye, 'hello' is the root (World) domain, and 'goodbye' is the subdomain. This contrasts with traditional Web2/Web3 domain naming, where for the domain hello.goodbye.com, 'hello' — the leftmost item —- is the 3rd level domain or a subdomain, 'goodbye' is the 2nd level domain or a subdomain and .com the rightmost, is the root domain. ZNS has chosen to utilize this left-right top-down hierarchy for several reasons. First, it is the standard organizational paradigm for the world's digital files systems (Windows, Linux, Mac, etc) and analog indexing systems (Dewey Decimal or Universal Decimal Classification, Harvard-Yenching Classification, etc.). Second, it mimics conventional left-right top-down word order and directional reading practiced by the majority of the world and world's languages. Third, and finally, it accommodates infinite extensibility of ZNS domains (a subdomain of a subdomain of a subdomain of a subdomain...) in a manner that is intuitively parseable by humans. This feature is glaringly missing from current Web2/Web3 domain nomenclature. In this sense, while still well-suited for denominating domain hierarchies, ZNS also accommodates broader routing and indexing purposes, making the product far more versatile than existing Web3 domain options today.

Description: The documentation denotes the structure `wilder.wapes` , indicating that `wilder` is the root domain and `wapes` is its subdomain. This portrayal is in contrast with the traditional way domains and subdomains are represented. Typically, given the full string `wilder.wapes` , `wilder` would be the subdomain, and `wapes` would be the domain.

Recommendation: If the described structure is a mistake by the team, it would be advisable to correct the documentation for clarity and consistency. However, if the team intentionally chose this representation, it would be helpful to include a clear note in the documentation explaining the choice and highlighting the differences from conventional Web2 domain/subdomain structures to avoid potential confusion among users and developers.

ZNS-14
Potential for Disrupted Access Control Due to Registry Mismatch

● Informational ⓘ Acknowledged

Update

The client acknowledged the issue and provided the following explanation:

> The recommendation provided in the report was one of the original thoughts when designing the system. But we chose to not implement it this way to avoid additional gas cost for an external call every time we need to read registry address in an AC modifier or anywhere else. Some points to consider: Any function or modifier using the registry will incur an additional gas increase to read registry from an external contract. ZNSRegistry is not expected to be changed, since it is a crucial contract to the system and will hold an extensive amount of data without which zNS will not function. ZNSRegistry is made a proxy, meaning if any changes to the contract logic are needed, they will be added through a proxy upgrade flow that will keep the original contract address, removing the need to update it on any external contract. The setRegistry() function has been added to each contract as a way to fix crucial failures or errors in the system, which may trigger a redeploy of the ZNSRegistry contract. This would happen only in the case of some very significant issue with the system and is not expected to happen in other situations. So it is not expected to ever call these functions unless some major problem is requiring a new Registry contract. If we get to the need of changing registry address on zNS contracts, this would mean that we lost or are losing the most crucial system storage where all the active domain data is stored. If we get to this situation, it may require an upgrade of other contracts in the system, at which point we would introduce a multicall like function to RootRegistrar that could call all setRegistry() functions on all contracts in a single tx to re-set registry address everywhere.

File(s) affected: `ZNSCurvePricer.sol` , `ZNSRootRegistrar.sol` , `ZNSSubRegistrar.sol` , `ZNSAddressResolver.sol` , `ZNSTreasury.sol` , `ZNSDomainToken.sol`

Description: The protocol contracts contain a setter function to update the registry contract. If the address of the registry contract is updated and missed for one of the contracts, it can lead to a mismatched registry across the different contracts. This mismatch is concerning since the registry plays a pivotal role in determining privileged roles, domain ownership, and associated operations. Consequently, if the registry is mismatched, it could disrupt the control and permissions system, affecting which functions are accessible when updating configurations. This is an information issue because it is a limitation of the design choice.

Recommendation: To mitigate potential mismatches and maintain synchronization across contracts, consider implementing a central DNS-like contract that provides the latest registry address. All contracts should reference this central contract for the current registry address.

ZNS-15 Privileged Roles

● Informational Fixed

Update

The client fixed the issue in `a0fe0d7b53eb138b7fed49caf2a66e838fd5a592` by documenting the roles in the repository's documentation. We recommend including the new admin-only functions such `addResolverType()` and `deleteResolverType()` as introduced during the fix review.

Description: All privileged permissions should be clearly documented for users. The privileged permissions of the protocol contract are documented below.

The following are cases of `GOVERNOR_ROLE` privileges:

- The `UUPS` function `upgradeToAndCall()` allows governors to update the implementation used and invoke a call in upgradeable contracts.
- The `UUPS` function `upgradeTo()` allows governors to update the implementation used in upgradeable contracts.
- The governors can grant `GOVERNOR_ROLE` to addresses.
- The governors can grant `ADMIN_ROLE` to addresses.
- The governors can grant any role to any address through the function `ZNSAccessController.setRoleAdmin()`.

The following are cases of `ADMIN_ROLE` privileges:

- The function `setRegistry()` allows admins to update the `registry` address for contracts inheriting `ARegistryWired`:
  - `ZNSCurvePricer`
  - `ZNSRootRegistrar`
  - `ZNSSubRegistrar`
  - `ZNSAddressResolver`
  - `ZNSDomainToken`
  - `ZNSTreasury`
- The function `ZNSRootRegister.setRootPricer()` allows admins to update the `pricer` contract used to determine pricing for root domains.
- The function `ZNSRootRegister.setTreasury()` allows admins to update the `ZNSTreasury` contract used to store protocol fees and staked funds.
- The function `ZNSRootRegister.setDomainToken()` allows admins to update the domain token contract used to validate domain ownership.
- The function `ZNSRootRegister.setSubRegistrar()` allows admins to update the subdomain registrar contract.
- The function `ZNSRootRegister.setAddressResolver()` allows admins to update the root domain resolver.
- The admins can grant `REGISTRAR_ROLE` to addresses.
- (Fix Review) The function `ZNSRegistry.addResolverType()` allows admins to add new resolver types to the mapping of resolver types and addresses.
- (Fix Review) The function `ZNSRegistry.deleteResolverType()` allows admins to delete resolver types to the mapping of resolver types and addresses.

The following are cases of `REGISTRAR_ROLE` privileges:

- The function `ZNSRootRegister.coreRegister()` allows registrars to register domains.
- The function `ZNSRegistry.createDomainRecord()` allows registrars to register domain records which track ownership and address solver.
- The function `ZNSDomainToken.register()` allows registrars to mint tokens which are used to validate domain ownership.
- The function `ZNSDomainToken.revoke()` allows registrars to burn tokens to revoke domain ownership.
- The function `ZNSTreasury.stakeForDomain()` allows registrars to process registration fee to beneficiaries and stake domain funds in the treasury. The staked funds are returned to the domain owner when the domain is revoked.
- The function `ZNSTreasury.unstakeForDomain()` allows registrars to unstake domain registration funds in the treasury during the domain revocation process.
- The function `ZNSTreasury.processDirectPayment()` allows registrars to process registration fees to beneficiaries directly.

Based on documentation and discussion with the client, the `REGISTRAR_ROLE` is reserved for contracts `ZNSRootRegistrar` and `ZNSSubRegistrar` only.

The role `EXECUTOR_ROLE` does not have any privileges. Based on the code documentation, this role may be used for future implementations.

Recommendation: Clarify the impact of these privileged actions on the end-users via publicly facing documentation.

ZNS-16 Upgradability

● Informational ⓘ Acknowledged

Update

The client acknowledged the issue and provided the following explanation:

> Most of the contracts in zNS (except ZNSAccessController) are upgradeable UUPS Proxies. We understand the limitations it entails, so please consider the following: We decided to go with upgradeable pattern for zNS to ensure we are able to fix any potential problems that may arise in the system with usage. We also wanted to have an ability to evolve system over time with the help of user feedback and possibly add features that our users might find necessary. Upgradability of contracts will be managed by the Zero DAO, which will directly control all aspects of zNS over time. The control will start with a Zero 6 of 12 multisig and slowly evolve into a full fledged DAO which will gain more and more control over time as the system logic finalizes. All upgradeable proxies are of UUPS kind. One of the main reasons to use UUPS here is to introduce a way to remove upgradability with time after the system is fully finalized and proven to be bug free or if the Zero DAO decides to do it themselves. Since Zero is an open-source platform, all smart contract changes coming with proxy upgrades will be public and available for anyone to see and analyze. In addition to that, any significant logic changes to existing contracts or addition of new contracts will be audited before they are deployed to Ethereum mainnet. When the DAO is fully functional, only the DAO will be able to approve a mainnet contract upgrade.

Description: While upgradeability is not a vulnerability in itself, users of the ZNS protocol should be aware that many contracts in the system can be upgraded at any time. This carries certain risks, as upgrading contracts is a delicate operation that can introduce bugs, regressions, and other undesirable code. It is important to note that this audit does not guarantee the behavior of future contracts that the protocol may be upgraded to.

Here is a list of contracts that have been explicitly marked as upgradeable (please note that the list might not be exhaustive):

```
▶ grep -rl --exclude="I*" --exclude-dir="mocks" "function initialize" contracts
contracts/token/ZNSDomainToken.sol
contracts/registrar/ZNSSubRegistrar.sol
contracts/registrar/ZNSRootRegistrar.sol
contracts/price/ZNSFixedPricer.sol
contracts/price/ZNSCurvePricer.sol
contracts/registry/ZNSRegistry.sol
contracts/treasury/ZNSTreasury.sol
contracts/resolver/ZNSAddressResolver.sol
```

Recommendation: Document and communicate clearly to the ZNS users that certain contracts in the protocol are upgradeable. Additionally, consider documenting the risks associated with contract upgrades and the mitigation strategies that will be followed to prevent the introduction of bugs or other undesirable code.

ZNS-17 Unlocked Pragma

● Informational ⓘ Fixed

Update

The client fixed the issue in `fc47e3de2198fdc557769249e7bb4b46415adba3` by locking the pragma version to `0.8.18`.

Related Issue(s): SWC-103

Description: Every Solidity file specifies in the header a version number of the format `pragma solidity (^)0.8.*`. The caret (`^`) before the version number implies an unlocked pragma, meaning that the compiler will use the specified version *and above*, hence the term "unlocked".

Recommendation: For consistency and to prevent unexpected behavior in the future, we recommend removing the caret to lock the file onto a specific Solidity version.

ZNS-18 Domain Address Resolver Can Be Malicious

● Undetermined ⓘ Fixed

Update

The client addressed the issue in `a3acd342663c0301d5cf5749081ac04e7934dfce` by introducing a `resolvers` mapping in the `ZNSRegistry`, linking resolver types to their approved addresses. When registering or updating a domain's resolver, users are required to provide a `resolverType`. The protocol then utilizes the approved resolver address corresponding to this type. Furthermore, only the protocol admin has the authority to add or remove resolver types and addresses, which mitigates the risk of malicious resolver introductions into the protocol.

In the fix provided, it is possible to delete a non-existing resolver type and have an event emitted. We recommend checking whether the resolver type exists before emitting the event.

Similarly, when updating the resolver type, it is also possible to set the type to a non-existing resolver. We recommend communicating this risk in the user-facing documentation, or validating that the string input corresponds to an existing resolver type.

File(s) affected: `ZNSRegistry.sol`

Description: The `ZNSSubRegistrar` contract provides functionality for a domain owner/operator to update the resolver contract for a specific domain hash via the `ZNSRegistry.updateDomainResolver()` or `ZNSRegistry.updateDomainRecord()` function. If manipulated by a malicious owner or operator, they could redirect requests to a malevolent resolver that hijacks the intended flow of external contracts trying to resolve a ZNS domain address. Although the integrity of the protocol is not directly compromised, any third-party systems relying on the external resolver are susceptible to unexpected and potentially malicious behaviors.

If users adhere to the `IZNSAddressResolver` interface, which mandates that `getAddress()` is a view function, the adverse potential of this issue is limited but still possible. The designation of `getAddress()` as a view function means the contract resolving the domain will use the `STATICCALL` opcode, which ensures that the state cannot be altered during the call. However, external protocols are not guaranteed to use `IZNSAddressResolver`.

Recommendation: Consider enforcing stricter control over resolver modifications by implementing a whitelisting mechanism or an approval process for known trustworthy resolver contracts. This would minimize the chances of introducing malevolent resolvers into the system. Moreover, it is crucial to transparently communicate these inherent risks to third-party developers or systems interfacing with this protocol. They should be advised to incorporate additional safeguards when using the resolver to retrieve ZNS domain addresses.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Code Documentation

1. The code documentation for function `ZNSCurvePricer.setPriceConfig()` incorrectly states that only the admin can access the setter function. However, the function is only accessible by the domain owner or operators.
2. The code documentation for contract `ZNSCurverPricer` incorrectly states that only the admin can access the setter functions. This is incorrect because only the domain owner or operators can access the price configuration setters.
3. `IDistributionConfig.sol` states that `LOCKED: The parent domain is locked which mean no subdomains can be registered`, however, `ZNSSubRegistrar.registerSubdomain()` allows the registering of subdomains for locked domains in case the caller passes the `isOwnerOrOperator` check. Clarify this in the documentation if this is intended behavior.

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

- `ca1...d19 ./contracts/resolver/IZNSAddressResolver.sol`
- `a8d...695 ./contracts/resolver/ZNSAddressResolver.sol`
- `a90...1ad ./contracts/treasury/IZNSTreasury.sol`
- `2a9...7b0 ./contracts/treasury/ZNSTreasury.sol`

- `ef8...155 ./contracts/registry/IZNSRegistry.sol`
- `755...f63 ./contracts/registry/ARegistryWired.sol`
- `b3a...ce1 ./contracts/registry/ZNSRegistry.sol`
- `fe8...be6 ./contracts/price/IZNSCurvePricer.sol`
- `360...3d2 ./contracts/price/IZNSFixedPricer.sol`
- `62c...532 ./contracts/price/ZNSFixedPricer.sol`
- `0b5...36c ./contracts/price/ZNSCurvePricer.sol`
- `7e2...ee9 ./contracts/utils/StringUtils.sol`
- `94c...d8e ./contracts/oz-proxies/TransparentUpgradeableProxyAcc.sol`
- `359...e2f ./contracts/oz-proxies/ERC1967ProxyAcc.sol`
- `0b7...5c8 ./contracts/registrar/ZNSSubRegistrar.sol`
- `ee6...a44 ./contracts/registrar/IZNSSubRegistrar.sol`
- `c4d...f96 ./contracts/registrar/ZNSRootRegistrar.sol`
- `2ab...ff1 ./contracts/registrar/IZNSRootRegistrar.sol`
- `4b3...08e ./contracts/access/IZNSAccessController.sol`
- `747...f64 ./contracts/access/AAccessControlled.sol`
- `06b...f1e ./contracts/access/ZNSAccessController.sol`
- `bc7...410 ./contracts/access/ZNSRoles.sol`
- `5af...5ca ./contracts/types/IDistributionConfig.sol`
- `6a1...bde ./contracts/types/IZNSPricer.sol`
- `cf6...27d ./contracts/types/ICurvePriceConfig.sol`
- `cf5...c8a ./contracts/token/IZNSDomainToken.sol`
- `50b...d87 ./contracts/token/ZNSDomainToken.sol`

Tests

- `2c1...979 ./test/ZNSTreasury.test.ts`
- `a3e...250 ./test/ZNSAccessController.test.ts`
- `e6d...c79 ./test/ZNSAddressResolver.test.ts`
- `13a...c25 ./test/ZNSDomainToken.test.ts`
- `b06...3bf ./test/ZNSSubRegistrar.test.ts`
- `e0b...bca ./test/ZNSRootRegistrar.test.ts`
- `912...725 ./test/ZNSFixedPricer.test.ts`
- `292...d1b ./test/ZNSRegistry.test.ts`
- `cf1...915 ./test/ZNSCurvePricer.test.ts`
- `f95...266 ./test/helpers/hashing.ts`
- `ec5...520 ./test/helpers/validate-upgrade.ts`
- `512...bb7 ./test/helpers/errors.ts`
- `1b8...50a ./test/helpers/balances.ts`
- `a57...bc6 ./test/helpers/pricing.ts`
- `b16...2bd ./test/helpers/utils.ts`
- `cbd...e32 ./test/helpers/types.ts`
- `668...8da ./test/helpers/tokens.ts`
- `9f3...65d ./test/helpers/access.ts`
- `8d7...347 ./test/helpers/constants.ts`
- `5c1...86d ./test/helpers/events.ts`
- `93f...758 ./test/helpers/index.ts`
- `091...4a1 ./test/helpers/register-setup.ts`
- `934...070 ./test/helpers/flows/registration.ts`
- `88f...cfb ./test/helpers/deploy/mocks.ts`
- `ca0...c31 ./test/helpers/deploy/deploy-zns.ts`
- `b3c...354 ./test/gas/TransactionGasCosts.test.ts`

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- Slither v0.9.6

Steps taken to run the tools:

1. Install the Slither tool: `pip3 install slither-analyzer`
2. Run Slither from the project directory: `slither . --filter-paths "utils/StringUtils.sol|token/mocks|upgrade-test-mocks"`

Automated Analysis

Slither

We have executed slither and filtered the issues that were reported and incorporated the valid ones in the report. Please note that only issues related to the scope of the audit are reported.

Test Suite Results

```
  ZNSAccessController
    Initial Setup
      ✔ Should assign GOVERNORs correctly
      ✔ Should assign ADMINs correctly
      ✔ Should revert when passing 0x0 address to assing 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 (56ms)
      ✔ GOVERNOR_ROLE should be able to grant ADMIN_ROLE
      ✔ GOVERNOR_ROLE should be able to revoke ADMIN_ROLE (58ms)
      ✔ 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 revoke REGISTRAR_ROLE (82ms)
      ✔ GOVERNOR_ROLE should be able to assign new EXECUTOR_ROLE as admin for REGISTRAR_ROLE
      ✔ GOVERNOR_ROLE should be able to make himself a new EXECUTOR_ROLE's admin and assign this role to
anyone (52ms)
      ✔ 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
      ✔ #isGovernor() should return true for GOVERNOR_ROLE
      ✔ #isExecutor() should return true for EXECUTOR_ROLE (50ms)
      ✔ 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 EXECUTOR_ROLE

  ZNSAddressResolver
Warning: All subsequent Upgrades warnings will be silenced.

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

    ✔ 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 (58ms)
    ✔ Should revert when setRegistry() without ADMIN_ROLE
    ✔ Should setAccessController() correctly with ADMIN_ROLE (57ms)
    ✔ Should revert when setAccessController() without ADMIN_ROLE
    ✔ Should not allow non-owner address to setAddress
    ✔ Should allow owner to setAddress and emit event (65ms)
    ✔ Should allow operator to setAddress and emit event (113ms)
    ✔ Should allow REGISTRAR_ROLE to setAddress and emit event (102ms)
    ✔ Should resolve address correctly (64ms)
    ✔ 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 (65ms)
    UUPS
      ✔ Allows an authorized user to upgrade the contract (99ms)
      ✔ Fails to upgrade if the caller is not authorized (87ms)
      ✔ Verifies that variable values are not changed in the upgrade process (341ms)

  ZNSCurvePricer
    ✔ Should NOT let initialize the implementation contract
    ✔ Confirms values were initially set correctly
    #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 base price for domains that are equal to the base length
      ✔ Returns the base 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 (52ms)
      ✔ Returns a price for multiple lengths
      ✔ Can Price Names Longer Than 255 Characters (54ms)
      - Doesn't create price spikes with any valid combination of values (SLOW TEST, ONLY RUN LOCALLY)
    #setPriceConfig
      ✔ Can't price a name that has invalid characters
      ✔ Should set the config for any existing domain hash, including 0x0 (290ms)
      ✔ Should revert if setting a price config where spike is created at maxLength
      ✔ Cannot go below the set minPrice
      ✔ Should revert if called by anyone other than owner or operator
      ✔ Should emit PriceConfigSet event with correct parameters (140ms)
      ✔ Fails validation when maxPrice < minPrice
    #setMaxPrice
      ✔ Allows an authorized user to set the max price (83ms)
      ✔ Disallows an unauthorized user to set the max price
      ✔ Allows setting the max price to zero (68ms)
      ✔ Correctly sets max price (86ms)
      ✔ Should revert when setting maxPrice that causes a spike at maxLength
      ✔ Causes any length domain to have a price of 0 if the maxPrice is 0 (80ms)
      ✔ The price of a domain is modified relatively when the basePrice is changed (93ms)
    #setMinPrice
      ✔ Allows an authorized user to set the min price (148ms)
      ✔ Disallows an unauthorized user from setting the min price
      ✔ Allows setting to zero (171ms)
      ✔ Successfully sets the min price correctly (87ms)
      ✔ Causes any domain beyond the `maxLength` to always return `minPrice` (192ms)
      ✔ Should revert when setting minPrice that causes a spike at maxLength
    #setPrecisionMultiplier
      ✔ Allows an authorized user to set the precision multiplier (72ms)
      ✔ Disallows an unauthorized user from setting the precision multiplier
      ✔ Fails when setting to zero
      ✔ Successfuly sets the precision multiplier when above 0 (72ms)
      ✔ Verifies new prices are affected after changing the precision multiplier (78ms)
      ✔ Should revert when setting precisionMultiplier higher than 10^18
    #setBaseLength
      ✔ Allows an authorized user to set the base length (83ms)
      ✔ Disallows an unauthorized user to set the base length
      ✔ Allows setting the base length to zero (79ms)
      ✔ Always returns the minPrice if both baseLength and maxLength are their min values (163ms)
      ✔ Causes any length domain to cost the base fee when set to max length of 255 (97ms)
      ✔ Causes prices to adjust correctly when length is increased (99ms)
      ✔ Causes prices to adjust correctly when length is decreased (176ms)
      ✔ Returns the maxPrice whenever the baseLength is 0 (170ms)
      ✔ Adjusts prices correctly when setting base lengths to different values (79ms)
      ✔ Should revert when setting baseLength that causes a spike at maxLength
    #setMaxLength
      ✔ Allows an authorized user to set the max length (86ms)
      ✔ Disallows an unauthorized user to set the max length
      ✔ Allows setting the max length to zero (68ms)
      ✔ Still returns prices for domains within baseLength if the maxLength is zero (79ms)
      ✔ Should revert when setting maxLength that causes a spike at maxLength
    #setFeePercentage
```

```
      ✔ Successfully sets the fee percentage (72ms)
      ✔ Disallows an unauthorized user to set the fee percentage
      ✔ should revert when trying to set feePercentage higher than PERCENTAGE_BASIS
    #getRegistrationFee
      ✔ Successfully gets the fee for a price
    #setAccessController
      ✔ Successfully sets the access controller (62ms)
      ✔ Disallows an unauthorized user to set the access controller
      ✔ Disallows setting the access controller to the zero address
    #setRegistry
      ✔ Should successfully set the registry (65ms)
      ✔ Should NOT set the registry if called by anyone other than ADMIN_ROLE
    Events
      ✔ Emits MaxPriceSet (83ms)
      ✔ Emits BaseLengthSet (84ms)
    UUPS
      ✔ Allows an authorized user to upgrade the contract (108ms)
      ✔ Fails to upgrade if the caller is not authorized (87ms)
      ✔ Verifies that variable values are not changed in the upgrade process (440ms)

  ZNSDomainToken
    ✔ should initialize correctly
    ✔ should NOT initialize twice
    ✔ Should NOT let initialize the implementation contract
    External functions
      ✔ Should register (mint) the token if caller has REGISTRAR_ROLE (78ms)
      ✔ Should revert when registering (minting) if caller does not have REGISTRAR_ROLE
      ✔ Revokes a token (269ms)
    Require Statement Validation
      ✔ Only the registrar can call to register a token
      ✔ Only authorized can revoke a token (108ms)
      ✔ Should set access controller if caller has ADMIN_ROLE (60ms)
      ✔ 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 (150ms)
      ✔ should set and correctly retrieve royalty for a specific token (153ms)
      ✔ #setDefaultRoyalty() should revert if called by anyone other than ADMIN_ROLE
      ✔ #setTokenRoyalty() should revert if called by anyone other than ADMIN_ROLE (96ms)
    Token URIs
      ✔ should support individual tokenURIs (79ms)
      ✔ should support baseURI method with tokenURI as 0 (143ms)
      ✔ should support baseURI + tokenURI concatenation if both are set correctly (139ms)
      ✔ should return WRONG URI if both baseURI and tokenURI are set as separate links (149ms)
      ✔ should be able to switch from tokenURI to baseURI if tokenURI is deleted (228ms)
      ✔ #setTokenURI() should set tokenURI correctly (235ms)
      ✔ #setTokenURI() should revert if called by anyone other than ADMIN_ROLE (105ms)
      ✔ #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 (111ms)
      ✔ Verifies that variable values are not changed in the upgrade process (413ms)
      ✔ Fails to upgrade if the caller is not authorized (78ms)

  ZNSFixedPricer
    ✔ should deploy with correct parameters
    ✔ should NOT initialize twice
    ✔ Should NOT let initialize the implementation contract
    ✔ should set config for 0x0 hash (102ms)
    ✔ should not allow to be deployed by anyone other than ADMIN_ROLE
    ✔ #setPrice() should work correctly and emit #PriceSet event (76ms)
    ✔ #getPrice should return the correct price (76ms)
    ✔ #getPrice() should revert for invalid label when not skipping the label validation
    ✔ #getPriceAndFee() should return the correct price and fee (143ms)
    ✔ #setPrice() should revert if called by anyone other than domain owner
    ✔ #setFeePercentage() should set the fee correctly and emit #FeePercentageSet event (74ms)
    ✔ #setFeePercentage() should revert if called by anyone other than domain owner
    ✔ #setFeePercentage() should revert when trying to set feePercentage higher than PERCENTAGE_BASIS
    ✔ #setPriceConfig() should set the price config correctly and emit #PriceSet and #FeePercentageSet
events (100ms)
    ✔ #setPriceConfig() should revert if called by anyone other than domain owner or operator
    ✔ #setRegistry() should set the correct address (129ms)
    ✔ #setRegistry() should revert if called by anyone other than ADMIN_ROLE
    ✔ #setAccessController() should revert if called by anyone other than ADMIN_ROLE
    ✔ #setAccessController() should set the correct address (64ms)
    UUPS
      ✔ Allows an authorized user to upgrade the contract (220ms)
      ✔ Fails to upgrade if the caller is not authorized (94ms)
      ✔ Verifies that variable values are not changed in the upgrade process (435ms)

  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 (49ms)
    ✔ Should set access controller correctly with ADMIN_ROLE (64ms)
    ✔ Should revert when setting access controller without ADMIN_ROLE
    Audit fix with approved address resolvers
      ✔ Adds `address` resolver type and creates a record (71ms)
      ✔ 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 (130ms)
      ✔ Deletes a resolver type (118ms)
    Operator functionality
      ✔ Returns false when an operator is not allowed by an owner (46ms)
      ✔ Returns true when an operator is allowed by an owner (47ms)
      ✔ Returns false when an owner has not specified any operators
      ✔ Permits an allowed operator to update a domain record (159ms)
      ✔ Does not permit a disallowed operator to update a domain record (48ms)
      ✔ Does not permit an operator that's never been allowed to modify a record
      ✔ #isOperatorFor() should return true for an operator (46ms)
    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 (73ms)
      ✔ 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 (187ms)
      ✔ Cannot update a domain owner if the domain doesn't exist
      ✔ Can update a domain owner if the domain exists (120ms)
      ✔ Cannot update a domain resolver if the domain doesn't exist
      ✔ Can update a domain resolver if the domain exists (183ms)
      ✔ Cannot update a domain record if the owner is zero address (74ms)
      ✔ Can update a domain record if the resolver is zero address (126ms)
      ✔ cannot update a domain owner if owner is zero address
      ✔ Can update a domain resolver if resolver is zero address (52ms)
      ✔ Fails to update a record when caller is not owner or operator
      ✔ cannot update a domain's record if not an owner or operator (74ms)
      ✔ cannot update an domain's owner if not an owner or operator (74ms)
      ✔ cannot update a domain's resolver if not an owner or operator (75ms)
      ✔ Can delete record with REGISTRAR_ROLE (126ms)
      ✔ Cannot delete record without REGISTRAR_ROLE (92ms)
    Event emitters
      ✔ Emits an event when an operator is set (45ms)
      ✔ Emits events when a new domain is created (85ms)
      ✔ Emits an event when an existing domain is updated (128ms)
      ✔ Emits an event when a domain's owner is updated (114ms)
      ✔ Emits an event when a domain's resolver is updated (191ms)
      ✔ Emits an event when a domain record is deleted (58ms)
    UUPS
      ✔ Allows an authorized user to upgrade successfully (107ms)
      ✔ Fails when an unauthorized account tries to call to upgrade (56ms)
      ✔ Verifies that variable values are not changed in the upgrade process (387ms)
  ZNSRootRegistrar
    ✔ Gas tests (1106ms)
    ✔ Should NOT let initialize the implementation contract
    ✔ Allows transfer of 0x0 domain ownership after deployment (48ms)
    ✔ Confirms a new 0x0 owner can modify the configs in the treasury and curve pricer (265ms)
    ✔ 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
      ✔ #isOwnerOf() returns correct bools (630ms)
      ✔ #setSubRegistrar() should revert if called by address without ADMIN_ROLE
      ✔ #setSubRegistrar() should set the correct address (201ms)
      ✔ #setSubRegistrar() should NOT set the address to zero address
    Registers a root domain
      ✔ Can NOT register a TLD with an empty name
      ✔ Can register a TLD with characters [a-z0-9-] (1428ms)
      ✔ Fails for domains that use any invalid character
      ✔ Successfully registers a domain without a resolver or resolver content and fires a
#DomainRegistered event (456ms)
      ✔ Successfully registers a domain with distrConfig and adds it to state properly (542ms)
      ✔ Stakes and saves the correct amount and token, takes the correct fee and sends fee to Zero Vault
(676ms)
      ✔ Sets the correct data in Registry (590ms)
      ✔ Fails when the user does not have enough funds (105ms)
      ✔ Disallows creation of a duplicate domain (597ms)
      ✔ Successfully registers a domain without resolver content (493ms)
      ✔ Records the correct domain hash (555ms)
      ✔ Creates and finds the correct tokenId (736ms)
      ✔ Resolves the correct address from the domain (640ms)
      ✔ Should NOT charge any tokens if price and/or stake fee is 0 (415ms)
    Reclaiming Domains
      ✔ Can reclaim name/stake if Token is owned (709ms)
      ✔ Reclaiming domain token emits DomainReclaimed event (824ms)
      ✔ Cannot reclaim name/stake if token is not owned (547ms)
      ✔ Cannot reclaim if domain does not exist
      ✔ Domain Token can be reclaimed, transferred, and then reclaimed again (962ms)
      ✔ Can revoke and unstake after reclaiming (1082ms)
    Revoking Domains
      ✔ Revokes a Top level Domain, locks distribution and removes mintlist (1294ms)
      ✔ Cannot revoke a domain that doesnt exist
      ✔ Revoking domain unstakes (960ms)
      ✔ Cannot revoke if Name is owned by another user (579ms)
      ✔ No one can revoke if Token and Name have different owners (784ms)
      ✔ After domain has been revoked, an old operator can NOT access Registry (1026ms)
    State Setters
      #setAccessController
        ✔ Should set AccessController and fire AccessControllerSet event (72ms)
        ✔ Should revert if not called by ADMIN
        ✔ Should revert if new AccessController is address zero
      #setRegistry
        ✔ Should set ZNSRegistry and fire RegistrySet event (65ms)
        ✔ Should revert if not called by ADMIN
        ✔ Should revert if ZNSRegistry is address zero
      #setTreasury
        ✔ Should set Treasury and fire TreasurySet event (69ms)
        ✔ Should revert if not called by ADMIN
        ✔ Should revert if Treasury is address zero
      #setDomainToken
        ✔ Should set DomainToken and fire DomainTokenSet event (67ms)
        ✔ Should revert if not called by ADMIN
        ✔ Should revert if DomainToken is address zero
    UUPS
      ✔ Allows an authorized user to upgrade the contract (129ms)
      ✔ Fails to upgrade when an unauthorized users calls (73ms)
      ✔ Verifies that variable values are not changed in the upgrade process (1379ms)

  ZNSSubRegistrar
    Single Subdomain Registration
      ✔ should revert when trying to register a subdomain before parent has set it's config with
FixedPricer (720ms)
      ✔ should revert when trying to register a subdomain before parent has set it's config with
```

```
CurvePricer (718ms)
      ✔ should register subdomain with the correct tokenURI assigned to the domain token minted (938ms)
      ✔ Can register a subdomain with characters [a-z0-9] (818ms)
      ✔ 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 (935ms)
      ✔ should register subdomain with a label length of 100000 chars (178196ms)
      ✔ should revert when user has insufficient funds (254ms)
      ✔ should revert when user has insufficient allowance (95ms)
      ✔ should revert on payment when parent's beneficiary has not yet been set and when stakeFee is > 0
(3117ms)
    Operations within domain paths
      ✔ should register a path of 6 domains with different configs (104ms)
      ✔ should be able to register multiple domains under multiple levels for the same owner (8123ms)
      ✔ should revoke lvl 6 domain without refund, lock registration and remove mintlist (490ms)
      ✔ should revoke lvl 5 domain with refund (481ms)
      ✔ should register a new 2 lvl path at lvl 3 of the existing path (2214ms)
      ✔ should revoke lvl 3 domain (child) with refund after lvl 2 (parent) has been revoked (816ms)
      ✔ should let anyone register a previously revoked domain (2599ms)
      ✔ should NOT register a child (subdomain) under a parent (root domain) that has been revoked
(1311ms)
      ✔ should NOT register a child (subdomain) under a parent (subdomain) that has been revoked (356ms)
      ✔ should allow setting a new config and start distributing subdomain when registering a previously
revoked parent (3465ms)
    Token movements with different distr setups
      ✔ FixedPricer - StakePayment - stake fee - 5 decimals (2058ms)
      ✔ FixedPricer - StakePayment - no fee - 18 decimals (2173ms)
      ✔ FixedPricer - DirectPayment - no fee - 8 decimals (2320ms)
      ✔ CurvePricer - StakePayment - stake fee - 13 decimals (2281ms)
      ✔ CurvePricer - StakePayment - no fee - 2 decimals (2399ms)
      ✔ CurvePricer - DirectPayment - no fee - 18 decimals (2304ms)
      ✔ FixedPricer + DirectPayment with price = 0 - should NOT perform any transfers (1891ms)
      ✔ CurvePricer + DirectPayment with price = 0 - should NOT perform any transfers (2087ms)
      ✔ CurvePricer + StakePayment with price = 0 - should NOT perform any transfers (2069ms)
      ✔ FixedPricer + StakePayment with price = 0 - should NOT perform any transfers (1813ms)
      ✔ Setting price config in incorrect decimals triggers incorrect pricing (1993ms)
    Registration access
      ✔ should allow parent owner to register a subdomain under himself even if accessType is LOCKED
(621ms)
      ✔ should NOT allow others to register a domain when parent's accessType is LOCKED (760ms)
      ✔ should allow anyone to register a domain when parent's accessType is OPEN (614ms)
      ✔ should ONLY allow mintlisted addresses and NOT allow other ones to register a domain when
parent's accessType is MINTLIST (2098ms)
      ✔ #updateMintlistForDomain() should NOT allow setting if called by non-authorized account or
registrar (177ms)
      ✔ #updateMintlistForDomain() should fire a #MintlistUpdated event with correct params (145ms)
      ✔ should switch accessType for existing parent domain (937ms)
      ✔ should NOT allow to register subdomains under the parent that hasn't set up his distribution
config (722ms)
    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 both Name and Token (277ms)
      ✔ should allow to UPDATE domain data for subdomain (197ms)
      ✔ should TRANSFER ownership of a subdomain and let the receiver RECLAIM and then revoke with REFUND
(734ms)
      #setDistributionConfigForDomain()
        ✔ should re-set distribution config for an existing subdomain (302ms)
        ✔ should NOT allow to set distribution config for a non-authorized account
        ✔ should revert if pricerContract is passed as 0x0 address
      #setPricerContractForDomain()
        ✔ should re-set pricer contract for an existing subdomain (163ms)
        ✔ 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 (162ms)
        ✔ should NOT allow setting for non-authorized account
        ✔ should emit #PaymentTypeSet event with correct params (164ms)
    State setters
      ✔ Should NOT let initialize the implementation contract
      ✔ #setRootRegistrar() should set the new root registrar correctly and emit #RootRegistrarSet event
(94ms)
      ✔ #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 (75ms)
      ✔ #setRegistry() should not be callable by anyone other than ADMIN_ROLE
      ✔ #setAccessController() should not be callable by anyone other than ADMIN_ROLE
      ✔ #getAccessController() should return the correct access controller
      ✔ #setAccessController() should set the new access controller correctly and emit
#AccessControllerSet event (77ms)
    UUPS
      ✔ Allows an authorized user to upgrade the contract (129ms)
      ✔ Fails to upgrade if the caller is not authorized (76ms)
      ✔ Verifies that variable values are not changed in the upgrade process (1445ms)
      ✔ Allows to add more fields to the existing struct in a mapping (1131ms)

  ZNSTreasury
    ✔ Should initialize correctly
    ✔ should NOT initialize twice
    ✔ Should NOT let initialize the implementation contract
    ✔ should NOT deploy/initialize with 0x0 addresses as args (69ms)
    #stakeForDomain()
      ✔ Stakes the correct amount (286ms)
      ✔ Should revert if called from an address without REGISTRAR_ROLE
      ✔ Should fire StakeDeposited event with correct params (178ms)
    #unstakeForDomain()
      ✔ Unstakes the correct amount and saves the correct token (318ms)
      ✔ Should revert if called from an address without REGISTRAR_ROLE
    #processDirectPayment()
      ✔ should process payment correctly with paymentConfig set (589ms)
      ✔ should revert if paymentConfig not set
      ✔ should revert if called by anyone other than REGISTRAR_ROLE
      ✔ should emit DirectPaymentProcessed event with correct params (320ms)
    #setPaymentConfig(), BeneficiarySet and PaymentTokenSet
      ✔ should re-set payment config for an existing subdomain (92ms)
      ✔ 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 (81ms)
      ✔ 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 (81ms)
      ✔ Should revert when called by anyone other than owner or operator
      ✔ Should revert when paymentToken is address 0
    #setAccessController() and AccessControllerSet event
      ✔ Should set the correct address of Access Controller (70ms)
      ✔ Should revert when called from any address without ADMIN_ROLE
      ✔ Should revert when accessController is address 0
    #setRegistry() and RegistrySet event
      ✔ Should set the correct address of Registry (71ms)
      ✔ 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 (122ms)
      ✔ Fails when an unauthorized user tries to upgrade the contract (62ms)
      ✔ Verifies that variable values are not changed in the upgrade process (594ms)

  Transaction Gas Costs Test

      Root Domain Price:
        Gas Used: 461730
        Gas Diff: 30922

    ✔ Root Domain Price (693ms)

      Subdomain Price:
        Gas Used: 462627
        Gas Diff: 36311

    ✔ Subdomain Price (777ms)
  364 passing (11m)
  1 pending
```

Code Coverage

The code coverage was generated for using `yarn coverage` and `.solcover.js` provided below.

```
module.exports = {
  skipFiles: [
    'utils/StringUtils.sol',
    'token/mocks',
    'upgrade-test-mocks'
  ]
};
```

The testsuite coverage is high quality. However, we recommend adding a test case for each high and medium issues identified in this report to confirm fixes.

Fix review: The client provided test cases for each fix to ensure the effectiveness of the fixes.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
access/	100	100	100	100	
AAccessControlled.sol	100	100	100	100	
IZNSAccessController.sol	100	100	100	100	
ZNSAccessController.sol	100	100	100	100	
ZNSRoles.sol	100	100	100	100	
oz-**proxies/**	100	100	100	100	
ERC1967ProxyAcc.sol	100	100	100	100	
TransparentUpgradeableProxyAcc.sol	100	100	100	100	
price/	100	100	100	100	
IZNSCurvePricer.sol	100	100	100	100	
IZNSFixedPricer.sol	100	100	100	100	
ZNSCurvePricer.sol	100	100	100	100	
ZNSFixedPricer.sol	100	100	100	100	
registrar/	99.02	93.33	100	99.13	
IZNSRootRegistrar.sol	100	100	100	100	
IZNSSubRegistrar.sol	100	100	100	100	
ZNSRootRegistrar.sol	98.31	93.18	100	98.39	319
ZNSSubRegistrar.sol	100	93.48	100	100	

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
registry/	100	93.75	100	100	
ARegistryWired.sol	100	100	100	100	
IZNSRegistry.sol	100	100	100	100	
ZNSRegistry.sol	100	92.86	100	100	
resolver/	100	100	100	100	
IZNSAddressResolver.sol	100	100	100	100	
ZNSAddressResolver.sol	100	100	100	100	
token/	100	100	100	100	
IZNSDomainToken.sol	100	100	100	100	
ZNSDomainToken.sol	100	100	100	100	
treasury/	100	100	100	100	
IZNSTreasury.sol	100	100	100	100	
ZNSTreasury.sol	100	100	100	100	
types/	100	100	100	100	
ICurvePriceConfig.sol	100	100	100	100	
IDistributionConfig.sol	100	100	100	100	
IZNSPricer.sol	100	100	100	100	
All files	99.65	96.67	100	99.7	

Changelog

- 2023-10-23 - Initial report
- 2023-11-17 - Final report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by Quantstamp; however, Quantstamp does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication or other making available of the report to you by Quantstamp.

Notice of confidentiality

This report, including the content, data, and underlying methodologies, are subject to the confidentiality and feedback provisions in your agreement with Quantstamp. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Quantstamp.

Links to other websites

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Quantstamp. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites&aspo; owners. You agree that Quantstamp are not responsible for the content or operation of such web sites, and that Quantstamp shall have no liability to you or any other person or entity for the use of third-party web sites. Except as described below, a hyperlink from this web site to another web site does not imply or mean that Quantstamp endorses the content on that web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the report. Quantstamp assumes no responsibility for the use of third-party software on any website and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any output generated by such software.

Disclaimer

The review and this report are provided on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Quantstamp disclaims all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. You agree that your access and/or use of the report and other results of the review, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. 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. This report is based on the scope of materials and documentation provided for a limited review at the time provided. You acknowledge that Blockchain technology remains under development and is subject to unknown risks and flaws and, as such, the report may not be complete or inclusive of all vulnerabilities. The review is limited to the materials identified in the report and does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. The report does not indicate the endorsement by Quantstamp of any particular project or team, nor guarantee its security, and may not be represented as such. No third party is entitled to rely on the report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. Quantstamp does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or 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, or any related services and products, any hyperlinked websites, or any other websites or mobile applications, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third party. 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.