IMPORTANT INFORMATION FROM WISDOMTREE DIGITAL

WisdomTree Digital, which operates through one or more subsidiaries of WisdomTree Inc., is providing access to the security audit reports conducted by OpenZeppelin which follow this page. By accessing the accompanying reports, you hereby acknowledge and agree that you have read and understood the contents herein.

Statements in the accompanying reports regarding the design, implementation, integration, suitability or functionality of the token standards framework or one or more smart contracts, such as those related to compliance, fraud prevention, risk management, security, audits, regulatory adherence and operations, address activities, events or developments that generally will or may occur in the future, are based on certain assumptions of the audit firm, are not tailored to any particular entity and no assurances can be made regarding the foregoing, notwithstanding statements in the accompanying reports such as those related to ensuring, enabling, effectiveness, robustness or suitability of certain functionality, operations or otherwise. Actual implementation, integration, suitability or functionality are all highly dependent on potential technical implementation and an issuer's, intermediary's or other applicable market participants' specific considerations, which may include those related to domicile, current or future investor or user base, regulatory requirements and other regulatory considerations, operating, compliance or risk management processes or procedures, desired functionality, technological know-how or comfort, general expectations or predictions, all of which are subject to a number of uncertainties, may change, may not come to fruition or may not work as intended.

In addition, blockchain and related technology, including smart contracts, are a relatively new and untested technology. The findings in the accompanying reports should not be considered a comprehensive list of security or other issues, flaws or defects in the framework, system or codebase. Token frameworks and smart contracts are subject to vulnerabilities and coding errors and no assurance can be made regarding the token standards framework or smart contract suitability, functionality, quality or reliability, even after audit recommendations have been implemented. WisdomTree Digital expressly disclaims all liability arising from the use of this information or reliance on the information provided and to the fullest extent permitted by applicable law, all warranties, whether express or implied, are hereby disclaimed by WisdomTree Digital. Use of this information is at the sole risk of the user. WisdomTree Digital does not provide financial, legal, regulatory or other professional advice.

This disclosure statement and the accompanying reports cannot and do not disclose all the risks and other factors necessary to evaluate a potential product or service from WisdomTree Digital. Before you decide to purchase or use a product or service from WisdomTree Digital or its affiliates, carefully read the specific information regarding such product or service (e.g., prospectus or other disclosure document, as applicable), including applicable risk disclosures.

OpenZeppelin | security

WisdomTree Tokenization Audit

August 11, 2025

Table of Contents

Summary

Type	RWA Tokenization
Timeline	From 2025-07-01 To 2025-07-14
Languages	Solidity

Total Issues	30 (29 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	9 (8 resolved)
Notes & Additional Information	21 (21 resolved)

Scope

OpenZeppelin conducted an audit of the feature/clean-up branch of the wisdomtreeam/wisdomtree_tokenization repository at commit 20e3fff. Changes made to previous files were compared against the master branch at commit 8db3b5c.

In scope were the following files:

```
src/
├── common
│   └── interfaces
│   │   └── IBeacon.sol
├── oracles
│   ├── interfaces
│   │   └── IOracleInit.sol
│   └── whitelist-compliance
│       └── WhitelistComplianceOracle.sol
├── proxies
│   ├── Proxy.sol
│   └── Beacon.sol
└── tokens
    ├── ERC20
    │   ├── common
    │   │   └── BaseERC20.sol
    │   └── standards
    │       ├── ERC20BasicStandard.sol
    │       ├── ERC20ControlledStandard.sol
    │       ├── ERC20RevocableComplianceStandard.sol
    │       └── ERC20RevocableStandard.sol
    ├── ERC721
    │   ├── ERC721BasicToken.sol
    │   └── ERC721SoulboundToken.sol
    └── interfaces
        ├── erc721
        │   └── IERC721TokenInit.sol
        ├── erc721Soulbound
        │   └── IERC721SoulboundTokenInit.sol
        └── erc20
            ├── IERC20.sol
            ├── IERC20BatchBasic.sol
            ├── IERC20BatchClawback.sol
            ├── IERC20BatchFreeze.sol
            ├── IERC20Burnable.sol
            ├── IERC20Clawback.sol
            ├── IERC20ClawbackEvents.sol
            ├── IERC20Errors.sol
            ├── IERC20Events.sol
            ├── IERC20Freeze.sol
```

```
        ├── IERC20FreezeEvents.sol
        ├── IERC20Mintable.sol
        ├── IERC20Pausable.sol
        ├── IERC20PausableEvents.sol
        ├── IERC20Permit.sol
        ├── IERC20RevocableCompliance.sol
        ├── IERC20Token.sol
        └── IERC20WithRoles.sol
```

Executive Summary

OpenZeppelin was engaged by the WisdomTree Digital team to conduct a comprehensive audit of the changes they wanted to introduce to their implementations of the ERC-20 and ERC-721 standards. The changes were motivated by a desire to adapt to the new beacon proxy pattern and include the permit functionality in the ERC-20 contracts. The goal of the audit team was to identify potential vulnerabilities, verify adherence to best practices, and provide recommendations for enhancing the overall security and functionality of the contracts. Only the smart contracts intended to be deployed on-chain were reviewed, while any deployment scripts, configurations, mock contracts, or tests remained out of scope.

The system includes a Whitelist Oracle designed to authorize transfers for token standards (e.g., ERC-20) by verifying whether the recipient holds any tokens from a specified ERC-721 or ERC-1155 contract or is directly allowed by a custom `WhitelistOracle` contract. In addition, the system implements two versions of an ERC-721 token. The first, used by the Whitelist Compliance Oracle, is a non-transferable "soulbound" version that can be minted to a single address and never transferred, designed to be held by whitelisted accounts.

The second is a standard implementation that, although not part of the system, serves as a foundation for more complex token designs. Additionally, there is a set of ERC-20 contracts that extend the standard to add certain functionalities such as batching operations, whitelisting, freezing, pausing, and the ability for the admin to clawback tokens from any given address. The Whitelist Oracle, ERC-721, and ERC-20 contracts employ a Beacon proxy pattern which allows for changing the implementations while also allowing these contracts to change their Beacon address too, maximizing the flexibility due to their use case.

This audit engaged two full-time auditors performing a manual, line-by-line review of the entire codebase as well as the use of automated code analysis for vulnerability detection. In particular, the audit validated that:

- All functionality performs as intended.
- The changes made to the Beacon implementation were propagated correctly in the contracts using it.
- The newly introduced permit functionality follows the standard.
- The role management system follows the specifications proposed for them.
- External actors can only interact with the system as intended.
- Best practices have been followed for code style and documentation.

This audit identified 30 issues. In particular, recommendations towards following the ERC-20 standard in edge cases, improving the Beacon upgrade process to prevent unexpected contract type changes during upgrades, handling non-whitelisted users' allowances, and following proxy standards were made. The remainder of the issues raised were aimed at improving the readability and maintainability of the codebase, addressing minor inconsistencies and redundancies within the code, and reducing gas costs. Overall, the codebase was found to be well-structured and to follow the best practices.

System Overview

The in-scope codebase includes a library of ERC-20 smart contracts that extend the standard and build on top of each other. The changes reviewed include the following:

- The `ERC20BasicStandard` contract, which introduces batching logic to allow for minting, burning, and transferring to multiple addresses within the same transaction.
- The `ERC20ControlledStandard` contract, which allows the admin to pause the contract and freeze individual addresses.
- The `ERC20RevocableStandard` contract, which adds functionality for the admin to clawback funds from addresses (i.e., move tokens between addresses at will).
- The `ERC20RevocableComplianceStandard` contract, which introduces a whitelisting mechanism that only allows whitelisted users to receive tokens.
- A new permit functionality for the `ERC20ControlledStandard` and `ERC20RevocableComplianceStandard` contracts.

Contracts have been designed to be upgradeable via the Beacon Proxy pattern. In addition, they can also upgrade the Beacon Proxy they point at. These two upgrades are controlled by the `DEFAULT_ADMIN_ROLE`. The upgrades involve validating contract types to ensure correct beacon and implementation usage, as well as adding other metadata. Contracts relying on this beacon have also been updated to work with the new `Beacon` implementation.

Security Model and Trust Assumptions

The contracts under review grant the admin a variety of privileges to effectively manage the token. Because of the possible attack vectors enabled through these privileges, the off-chain security model for the privileged address is critical. Otherwise, a malicious attacker could exploit the system in a variety of ways (e.g., moving all the funds to a custom address, minting themselves a vast amount of tokens, burning user funds, etc).

The project uses custom, role-based access control that is based on OpenZeppelin `AccessControl` library. There are the following privileged roles in the system:

- `DEFAULT_ADMIN_ROLE`: This default administrator is able to upgrade the implementation contract and the beacon address.
- `DELEGATED_ADMIN_ROLE`: This role manages the `REGISTRAR_ROLE`.
- `REGISTRAR_ROLE`: This role is allowed to mint, burn, freeze, pause, and clawback tokens, and set the ERC-721 token's URI.
- `ISSUER_ROLE`: This role is assigned but unused.
- `INIT_OWNER`: This role is allowed to initialize the implementation contracts.

Integration Concerns

When integrating with other protocols using the tokens/contracts under review, the following concerns should be noted:

- The `BaseERC20` contract mostly complies with the ERC-20 standard, except when a self-transfer is made without any tokens.
- Batching methods do not have the same behavior in edge cases compared to the ERC-20 standard.
- Funds might be removed from an account balance due to the `clawback` functionality, a privilege that is granted to the respective admin by default. This means that protocol funds might be subject to change without user intervention.

Test Suite

During the audit, it was observed that there was satisfactory test coverage across the files in scope. However, the test suite would benefit from additional edge cases, negative test scenarios, and fuzz testing to improve statement and branch coverage. Currently, the ERC-20 contracts rely exclusively on basic unit tests, resulting in notably low branch coverage - as little as 11% in some instances. During the audit kickoff, the WisdomTree Digital team confirmed they were actively working to enhance test coverage across the codebase.

After the audit, the team added more fuzz and unit tests cases which increased the statement coverage from 80% to 90% and the branch coverage to 60% for the core components in scope.

Low Severity

L-01 Enum Elements Cast to `Uint8`

In the `Beacon` contract, the elements of the `ContractType` enum are often cast into `uint8` values. However, other projects might rely on the usage of the enum in their validation logic to assert that the right type is being used, and if the list of types grows or is changed, it might cause problems down the line.

To improve code readability and reduce possible integration conflicts, consider using the enum elements as they are instead of converting them into `uint8` values.

Update: *Resolved in pull request #32 at commit f89e539, pull request #55 at commit 9a335b8, pull request #56 at commit acd565c, pull request #61 at commit 9878f4a, and pull request #66 at commit b405cf1. The `ContractType` `enum` has been refactored into the `ContractType.sol` file which is then imported in places where this enum is being used. Similarly, the `TokenType` elements have been refactored to use the `TokenType.sol` file.*

L-02 User Cannot Revoke Allowance to an Address That Has Been Removed from the Whitelist

In the `ERC20RevocableComplianceStandard` contract, the `permit` function prevents the user from adjusting the allowance of an address that is not whitelisted. Setting the allowance to a non-zero value should be prevented, but existing allowances should be allowed to be set to 0 for those addresses. Otherwise, a non-whitelisted user who still has a non-zero allowance on behalf of another user would still be able to move funds out of that user's wallet if the destination address is whitelisted.

Consider allowing users to set the allowance of non-whitelisted users to 0 through the permit.

Update: *Resolved in pull request #27 at commit 323d753.*

L-03 `REGISTRAR_ROLE` Being Used to Mint Instead of `ISSUER_ROLE`

The `ISSUER_ROLE` is assigned but not used in the `BaseERC20`, `ERC721BasicToken`, and `ERC721SoulboundToken` contracts. The documentation states that the `ISSUER_ROLE` is meant to issue or mint new tokens. However, in the codebase, the `REGISTRAR_ROLE` role is the one performing this function along with other several sensitive actions.

Consider using the `ISSUER_ROLE` for the issuing/minting operations as stated by the documentation. Alternatively, consider removing the role entirely and likewise adapting the documentation. Doing so will help reduce code complexity and accurately reflect the current implementation.

Update: *Resolved in pull request #28 at commit bf3c360. The WisdomTree team has added more documentation stating that the `ISSUER_ROLE` should be treated as a placeholder for future functionality.*

L-04 One Implementation Could Represent Multiple Versions

In the `Beacon` contract, the `_setImplementation` function takes the new implementation address as input and checks whether it has the right interface. If so, the contract stores it, increases the version by 1 unit, and emits the upgraded event. However, the contract does not check if the new implementation matches the current one. This means that if the same implementation address is being used, the version will be increased on the same implementation, resulting in 2 different versions corresponding to the same implementation. This would also pollute the off-chain systems monitoring the implementation changes.

Consider restricting the upgrade if the new implementation matches the existing one. Additionally, since the relationship between the version and the implementation is not being tracked, even when fixed, alternate implementations could reproduce the same issue. Consider taking into account such a pattern to prevent having the same implementation with different version numbers.

Update: *Resolved in pull request #30 at commit 9086427, pull request #59 at commit 255f4c2, and pull request #68 at commit 831fa2a. The WisdomTree team removed the entire concept of implementation versioning in the `Beacon` contract. The team stated:*

> *Due to the limitations of this incremental version tracking pattern, we have decided to remove this mechanism from the Beacon contract in this audit round. We will redesign our approach and implement it for the next audit.*

L-05 Overridden Functionalities Do Not Re-Use Parent Implementation

Throughout the codebase, a pattern of re-implementing entirely overridden functions, even if most of the logic is duplicated, was identified:

- In the `BaseERC20` contract, the `supportsInterface` function defines all the interfaces that are meant to be used in the contract. `BaseERC20` is then inherited by other child contracts, such as the `ERC20BasicStandard` and `ERC20ControlledStandard` contracts, and their `supportsInterface` function overrides and re-declares all the supported interfaces from the `BaseERC20` contract and the new `IERC20BatchBasic` interface without properly extending them using the `super` keyword. This could increase maintenance costs when changes are made to the different contracts, with the possibility of the introduction of bugs by forgetting to update the downstream inheritance tree.
- The same happens with functionalities that extend a functionality but do not share similar checks and flows. For instance, in the `ERC20ControlledStandard` contract, the `approve` function shares most of the functionality with the one in the `BaseERC20` contract but adds the `_isFrozen` check. This extension could use the `super.approve` after checking if it is frozen, which will reduce the amount of code and the maintenance effort.

To reduce the amount of code that needs to be maintained and the possible introduction of mismatches after changes are made to any of the contracts in the inheritance chain, consider re-using code from parent contracts and only extending the newly added functionality by using the `super` keyword.

Update: *Resolved in pull request #33 at commit 116289c, pull request 57 at commit b8ec7fb, pull request #62 at commit be0e698, pull request #63 at commit 756fc25, and in pull request #65 at commit 4e3198b. The aforementioned functions along with some other ones have been refactored.*

L-06 Unreachable Implementation

In the `BaseERC20` contract, the `_transfer` function implements the logic to adjust the balances of the `from` and `to` addresses with the `value` input. In particular, the function splits the functionality depending on whether each address is the zero address or not. If it is so, the implementation makes adjustments to the `_TOTAL_SUPPLY_SLOT` slot on each end. This logic is meant to be used in circumstances such as minting or burning tokens.

However, for such circumstances, the contract implements internal functions. Consequently, and also because any other overridden `transfer` or `transferFrom` function from a child contract restricts the calls with these addresses being the zero address, the branches belonging to the `_transfer` function with zero addresses will never be able to be called.

In order to reduce the attack surface and reduce code maintenance, consider removing the aforementioned logic from the `_transfer` function.

Update: *Resolved in pull request #31 at commit 6868dcb.*

L-07 `BaseERC20` Not ERC-20 Compliant

The `BaseERC20` contract implements the basic functionalities described by the ERC-20 standard. Additionally, its `_transfer` function prevents the transfer of tokens to the same address (i.e., the `to` and `from` addresses cannot be equal). However, while the standard does not prevent this, it does state that "Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event". In such a situation, if the value to transfer from/to the same address is zero, the `BaseERC20` contract would be violating the ERC-20 standard and would become non-compliant. This could break compatibility with projects that might rely on that same origin/destination transfer and check the emitted event.

Similarly, in the `ERC20BasicStandard` contract, the `batchTransfer` function restricts the transfers to amounts over zero, having the same effect. Even though this method is not part of the ERC-20 standard, its purpose is to batch the operations contained in the standard. This means that their behavior should mimic that of the standard `transfer` function, especially when it comes to handling batches of a single element.

To comply with the ERC-20 standard, consider allowing transfers in the aforementioned cases.

Update: *Resolved in pull request #26 at commit 7444fcc.*

L-08 Beacon Upgrade Can Skip Type Check and Use Wrong Type

Throughout the codebase, different contracts that are meant to use a `Beacon` proxy use a similar pattern to validate the new address and update the internal storage. In particular, the `WhitelistComplianceOracle`, the `BaseERC20`, the `ERC721BasicToken`, and the `ERC721SoulboundToken` contracts implement the `upgradeBeaconToAndCall` function to achieve this. To validate the compatibility of the new `Beacon` contract, the `upgradeBeaconToAndCall` function uses a couple of `try-catch` statements that skip the check if either the old or new `Beacon` contracts do not implement the `contractType` method, which otherwise should match between them to proceed.

This means that if the upgrade is done from `Beacon_A` (that has the `contractType` method) to `Beacon_B` (which does not have it), the next upgrade after `Beacon_B`, `Beacon_C`, could have an erroneous `ContractType` and still pass the upgrade, locking the wrong type for the respective contract until 2 other upgrades can be made, as there is no proper enforcement between the `Beacon` and the respective contract using it.

Even though the `upgradeBeaconToAndCall` function is guarded by the `DEFAULT_ADMIN_ROLE`, consider changing this pattern in all the affected contracts to prevent this situation from happening by enforcing the right type relationship between the contracts and `Beacon`s. Additionally, consider documenting the reasons why a `Beacon` contract would skip the successful implementation and call of such a function.

Update: *Resolved in pull request #29 at commit 77700bb. The `Beacon` implementations must have the `contractType` method, and the current and new implementations must match their type. It is worth noting that if an already deployed `Beacon` contract does not implement such a method, the `Beacon` address upgrade will always revert.*

L-09 Namespaced Storage Layout Does Not Follow The Standard

For the namespaced storage layout, there are two standards that specify the way storage locations should be calculated. In case of normal state variables, ERC-7201 specifies the storage address as `keccak256(abi.encode(uint256(keccak256(bytes(id))) -`

`1)) & ~bytes32(uint256(0xff))`. The standard should be used for the following storage instances:

- In the `WhitelistComplianceOracle` contract:
  - `_ORACLE_NAME_SLOT`
  - `_ORACLE_DESCRIPTION_SLOT`
  - `_INIT_OWNER_SLOT`
  - `_IS_INITIALIZED_SLOT`
  - `_ORACLE_ENABLED_SLOT`
  - `_MAX_CONTEXTS_SLOT`
  - `_CONTRACT_DATA_KEYS_SLOT`
  - `_CONTRACT_DATA_ADDRESS_SLOT`
  - `_CONTRACT_DATA_TYPE_SLOT`
  - `_CONTRACT_DATA_ID_SLOT`
- In the `Beacon` contract:
  - `_IMPLEMENTATION_VERSION_SLOT`
  - `_BEACON_NAME_SLOT`
  - `_CONTRACT_TYPE_SLOT`
- In the `BaseERC20` contract:
  - `_INIT_OWNER_SLOT`
  - `_TOTAL_SUPPLY_SLOT`
  - `_IS_INITIALIZED_SLOT`
  - `_DECIMALS_SLOT`
  - `_NAME_SLOT`
  - `_SYMBOL_SLOT`
  - `_BALANCES_SLOT`
  - `_ALLOWED_SLOT`
  - `_NONCES_SLOT`
  - `_DOMAIN_SEPERATOR_SLOT`
- In the `ERC20ControlledStandard` contract:
  - `_PAUSED_SLOT`
  - `_FROZEN_SLOT`

- In the `ERC20RevocableComplianceStandard` contract:
  - `_COMPLIANCE_ADDRESS_SLOT`

For the Beacon Proxy pattern, the slots for the implementation and beacon address should follow EIP-1967 to allow block explorers to properly display the proxies:

- `_BEACON_SLOT` in the `BaseERC20` contract
- `_IMPLEMENTATION_SLOT` in the `Beacon` contract
- `_BEACON_SLOT` in the `WhitelistComplianceOracle` contract
- `_BEACON_SLOT` in the `ERC721BasicToken` contract
- `_BEACON_SLOT` in the `ERC721SoulboundToken` contract

Consider updating the slot values to use the standards defined in the respective EIPs.

Update: *Acknowledged, will resolve. The WisdomTree team stated:*

> *Acknowledged. Given our current upgrade pattern on the mainnet, we are hesitant to make this change at this time to those existing logic contracts. The primary concern is over upgrading the current implementations and ensuring that the existing proxies continue to read the correct and expected slots without any storage collision risk or breaking those storage instances. There is no straightforward way to switch to a new storage layout on existing implementations through an upgrade without risking:*
>
> *1. **Storage Collisions**: Changing slot calculations could cause new slots to overlap with existing data.*
> *2. **Data Loss**: Existing data stored in current slots would become inaccessible.*
> *3. **Contract Breakage**: Proxies would read from incorrect storage locations. Our current storage layout is secure and collision-resistant.*
>
> *While we acknowledge that our current storage layout does not follow the ERC-7201 standard, changing the existing deployed contracts would pose unacceptable risks to user funds and contract functionality. Our current implementation is secure and functional, with no identified security vulnerabilities. **For future protocol deployments**, we commit to implementing the ERC-7201 standard. The pull request containing our future plans and approach can be found here.*

Notes & Additional Information

N-01 `ContractType` Defaults to `ERC20Token`

In the `Beacon` contract, the `ContractType` enum declares `ERC20Token` as the first enum value. This means that any uninitialized variable silently resolves as `ERC20Token`. This can mask deployment or upgrade mistakes by making an unset beacon appear valid.

Consider introducing an explicit placeholder such as NONE at index 0 and updating references accordingly. This will help ensure that accidental omissions surface instead of defaulting to an unintended `ERC20Token` configuration.

Update: *Resolved in pull request #43 at commit d8d9806.*

N-02 `README` Content Does Not Reflect the Project

The repository's root README.md file contains the default Foundry template content, offering no project-specific overview, setup guide, feature list, limitations, or usage examples. This hinders onboarding, forces integrators to reverse-engineer intent from code, and ultimately lowers perceived project quality.

Consider replacing the placeholder README file with a comprehensive document outlining the project's purpose, architecture, installation, environment variables, typical workflows, benefits, caveats, and links to further references.

Update: *Resolved in pull request #36 at commit 8f8e320.*

N-03 Missing Docstrings

Throughout the codebase, multiple instances of missing docstrings were identified:

- In the `IERC20RevocableCompliance` contract, the `ComplianceAdded` event
- In the `IERC20RevocableCompliance` contract, the `ComplianceRemoved` event

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #37 at commit eb15bd7.*

N-04 Incomplete Docstrings

Throughout the codebase, multiple instances of incomplete docstrings were identified:

- In the `ERC20ControlledStandard` contract, within the `approve` function, not all return values are documented.
- In the `ERC20ControlledStandard` contract, within the `isPaused` function, not all return values are documented.
- In the `ERC20RevocableComplianceStandard` contract, within the `getCompliance` function, not all return values are documented.
- In the `IERC20ClawbackEvents` interface, within the `Clawback` event, the `from`, `to`, and `tokens` parameters are not documented.
- In the `IERC20FreezeEvents` interface, within the `Freeze` event, the `account` parameter is not documented.
- In the `IERC20FreezeEvents` interface, within the `Unfreeze` event, the `account` parameter is not documented.
- In the `IERC20Pausable` interface, within the `isPaused` function, not all return values are documented.
- In the `IERC20Permit` interface, within the `permit` function, the `owner`, `spender`, `value`, `deadline`, `v`, `r`, and `s` parameters are not documented.
- In the `IERC20Permit` interface, within the `nonces` function, the `owner` parameter and multiple return values are not documented.
- In the `IERC20RevocableCompliance` interface, within the `getCompliance` function, not all return values are documented.
- In the `IBeacon` interface, within the `getContractType` function, the output number should be included in the documentation for the respective contract types, as is done in the implementation contract.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of a contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #38 at commit 3b353de and in pull request #43 at commit d8d9806.*

N-05 Unused Named Return Variables

Named return variables are a way to declare variables that are meant to be used within a function's body for the purpose of being returned as that function's output. They are an alternative to explicit in-line `return` statements.

Within the `BaseERC20` contract, multiple instances of unused named return variables were identified:

- The `result` return variable of the `balanceOf` function
- The `tokens` return variable of the `allowance` function

Consider either using or removing any unused named return variables.

Update: *Resolved in pull request #44 at commit 8def73e. Named returns have been removed.*

N-06 Use `calldata` Instead of `memory`

When dealing with the parameters of `external` functions, it is more gas-efficient to read their arguments directly from `calldata` instead of storing them to `memory`. `calldata` is a read-only region of memory that contains the arguments of incoming `external` function calls. This makes using `calldata` as the data location for such parameters cheaper and more efficient compared to `memory`. Thus, using `calldata` in such situations will generally save gas and improve the performance of a smart contract.

Throughout the codebase, multiple instances where function parameters should use `calldata` instead of `memory` were identified:

- In the `WhitelistComplianceOracle` contract, the `callData` parameter
- In the `Beacon` contract, the `newName` parameter
- In the `BaseERC20` contract, the `name_` parameter
- In the `BaseERC20` contract, the `symbol_` parameter
- In the `BaseERC20` contract, the `callData` parameter
- In the `ERC721BasicToken` contract, the `name_` parameter
- In the `ERC721BasicToken` contract, the `symbol_` parameter
- In the `ERC721BasicToken` contract, the `callData` parameter
- In the `ERC721BasicToken` contract, the `baseURI_` parameter

- In the `ERC721BasicToken` contract, the `tokenURI_` parameter
- In the `ERC721SoulboundToken` contract, the `name_` parameter
- In the `ERC721SoulboundToken` contract, the `symbol_` parameter
- In the `ERC721SoulboundToken` contract, the `callData` parameter
- In the `ERC721SoulboundToken` contract, the `baseURI_` parameter
- In the `ERC721SoulboundToken` contract, the `_tokenURI` parameter

Consider using `calldata` as the data location for the parameters of `external` functions to optimize gas usage.

Update: *Resolved in pull request #39 at commit a711380 and pull request #58 at commit 591a9df.*

N-07 Outdated Solidity Versions

Throughout the codebase, the version of Solidity used (`0.8.19`) is outdated. This can lead to vulnerabilities and unexpected behavior in contracts. It is important to keep the Solidity compiler version up to date.

Consider taking advantage of the latest Solidity version to improve the overall readability and security of the codebase as well as make use of the latest gas optimizations. For example, since `0.8.22`, there is no need to wrap loop increments in an `unchecked` block, and custom errors can be used in combination with the `require` statement instead of having to use the `if-clause` pattern.

Update: *Resolved in pull request #35 at commit 8d936dd. The Solidity version has been updated to `0.8.30`.*

N-08 Incremental Update Is Not Wrapped in an `unchecked` Block

Since Solidity version `0.8.0`, arithmetic operations include automatic checks for overflows and underflows, which increases gas costs. In scenarios where it is highly unlikely for a positively incrementing variable to overflow within a loop, using an `unchecked` block can optimize gas usage without compromising security. Before Solidity version `0.8.22`, developers manually implemented this optimization to bypass the additional overhead from automatic overflow checks.

Throughout the codebase, there are multiple opportunities where this optimization could be applied:

- The `++i` increment operation in the `removeContractAddress` function within the `WhitelistComplianceOracle` contract
- The `++i` increment operation in the `getContractAddresses` function within the `WhitelistComplianceOracle` contract
- The `++j` increment operation in the `getContractAddresses` function within the `WhitelistComplianceOracle` contract
- The `++j` increment operation in the `getContractAddresses` function within the `WhitelistComplianceOracle` contract
- The `uniqueCount++` increment operation in the `getContractAddresses` function within the `WhitelistComplianceOracle` contract
- The `++i` increment operation in the `getContractAddresses` function within the `WhitelistComplianceOracle` contract
- The `++i` increment operation in the `isAddressWhitelisted` function within the `WhitelistComplianceOracle` contract
- The `++i` increment operation in the `batchTransfer` function within the `ERC20BasicStandard` contract
- The `++i` increment operation in the `batchMint` function within the `ERC20BasicStandard` contract
- The `++i` increment operation in the `batchBurn` function within the `ERC20BasicStandard` contract
- The `++i` increment operation in the `batchTransfer` function within the `ERC20ControlledStandard` contract
- The `++i` increment operation in the `batchMint` function within the `ERC20ControlledStandard` contract
- The `++i` increment operation in the `batchBurn` function within the `ERC20ControlledStandard` contract
- The `++i` increment operation in the `batchFreeze` function within the `ERC20ControlledStandard` contract
- The `++i` increment operation in the `batchUnfreeze` function within the `ERC20ControlledStandard` contract
- The `++i` increment operation in the `batchTransfer` function within the `ERC20RevocableComplianceStandard` contract
- The `++i` increment operation in the `batchTransfer` function within the `ERC20RevocableComplianceStandard` contract
- The `++i` increment operation in the `batchMint` function within the `ERC20RevocableComplianceStandard` contract

- The `++i` increment operation in the `batchClawback` function within the `ERC20RevocableComplianceStandard` contract
- The `++i` increment operation in the `_areWhitelisted` function within the `ERC20RevocableComplianceStandard` contract
- The `++i` increment operation in the `batchClawback` function within the `ERC20RevocableStandard` contract
- The `++i` increment operation in the `batchSafeMint` function within the `ERC721BasicToken` contract
- The `StorageSlot.getUint256Slot(_NEXT_TOKEN_ID_SLOT).value++` increment operation in the `batchSafeMint` function within the `ERC721BasicToken` contract
- The `++i` increment operation in the `batchBurn` function within the `ERC721BasicToken` contract
- The `++i` increment operation in the `batchSafeMint` function within the `ERC721SoulboundToken` contract
- The `StorageSlot.getUint256Slot(_NEXT_TOKEN_ID_SLOT).value++` increment operation in the `batchSafeMint` function within the `ERC721SoulboundToken` contract
- The `++i` increment operation in the `batchBurn` function within the `ERC721SoulboundToken` contract

Consider either updating the pragma version to `0.8.22` to leverage automatic overflow check optimizations or wrapping incremental updates in an `unchecked` block to save gas.

Update: *Resolved in pull request #35 at commit 8d936dd. The Solidity version has been updated to* `0.8.30`*.*

N-09 Unused Error

In the `Beacon` contract, the `BeaconIncompatibleContractType` error is unused.

To improve the overall clarity, intentionality, and readability of the codebase, consider either using or removing any currently unused errors.

Update: *Resolved in pull request #45 at commit f2bca5e.*

N-10 Unused Functions With `internal` or `private` Visibility

Throughout the codebase, multiple instances of unused functions were identified:

- The `_increaseBalance` function in the `ERC721BasicToken` contract
- The `_increaseBalance` function in the `ERC721SoulboundToken` contract

To improve the overall clarity, intentionality, and readability of the codebase, consider either using or removing any currently unused functions.

Update: *Resolved in pull request #46 at commit 94ca6d9.*

N-11 Unused Import

The `IERC20Token` interface import in the `BaseERC20` contract is unused.

Consider removing unused imports to improve the overall clarity and maintainability of the codebase.

Update: *Resolved in pull request #47 at commit 5bf6d3e.*

N-12 Unspecific Custom Errors

In the `WhitelistComplianceOracle` contract, the `upgradeBeaconToAndCall` function uses the `OracleInvalidBeaconAddress` custom error to signal an issue in the different validation points. However, the `BaseERC20` contract, which also has a similar `upgradeBeaconToAndCall` function, uses more descriptive custom errors to describe the reason for the reversion.

This not only introduces inconsistency throughout the codebase but can also result in failing upgrades. To improve code readability and the debugging of reversions, consider using descriptive custom errors everywhere, especially in places with similar functionalities. Additionally, consider abstracting these custom errors in a different library so that they can be imported into different contracts.

Update: *Resolved in pull request #48 at commit ba33e30, pull request #64 at commit 8faddd7, and pull request #67 at commit b13e1f3. Beacon-upgradeable contracts now inherit more descriptive errors from the `IBeaconUpgradeErrors` interface. Old and now unused errors have been removed.*

N-13 Event Emissions Can Confuse Off-Chain Services

In the codebase, there are a few admin-protected functions that set sensitive addresses or values. Currently, these functions accept input values without verifying whether the new value differs from the current one, emitting events which suggest that the variable has changed its value, creating confusion for off-chain clients monitoring the event.

For instance, the `upgradeBeaconToAndCall` function from the `BaseERC20` contract accepts passing the same value for the `newBeacon` input as the existing one, which triggers the `BeaconChanged` event.

Consider adding a check to revert the execution if the value being set is identical to the existing one.

Update: *Resolved in pull request #49 at commit 81afe2f.*

N-14 Unused Events

The `Mint` event in the `IERC20Mintable` interface is not being used, as the minting functionality needs to use the `Transfer` event instead.

To improve the overall clarity, intentionality, and readability of the codebase, consider removing any unused events.

Update: *Resolved in pull request #50 at commit 59e62c8.*

N-15 Misleading Parameter Names

Throughout the codebase, multiple instances of misleading parameter names were identified:

- In the `BaseERC20` contract, the `totalSupply_` name suggests it will set the total supply for the ERC-20 contract. However, this variable represents the initial supply minted to a given address. The admin is still able to mint additional tokens using the mint function, so naming it to `initialSupply_` would reflect better its purpose.
- In the `BaseERC20` contract, the `tokens` parameter should be renamed to `amount`.
- In the `ERC20RevocableStandard` contract, the `value` parameter should be renamed to `amount`.

- In the `ERC20ControlledStandard` contract, the `tokens` parameter should be renamed to `amount`.

Consider addressing the aforementioned instances of misleading parameter names to improve the clarity and maintainability of the codebase.

Update: *Resolved in pull request #40 at commit 314d9c0.*

N-16 Instances of Docstrings Pointing to the Wrong Interface

Throughout the codebase, multiple instances of docstrings pointing to an `ERC20Batch` interface file that does not exist were identified:

- In the `ERC20BasicStandard` contract, the `batchTransfer`, `batchMint`, and `batchBurn` functions.
- In the `ERC20ControlledStandard` contract, the `batchTransfer`, `batchMint`, `batchBurn`, `batchFreeze`, and `batchUnfreeze` functions.
- In the `ERC20RevocableStandard` contract, the `batchClawback` function.
- In the `ERC20RevocableComplianceStandard` contract, the `batchTransfer`, `batchMint`, and `batchClawback` functions.

Consider updating the docstrings for the aforementioned functions so that they point to the `IERC20BatchBasic` interface.

Update: *Resolved in pull request #41 at commit 98b5348.*

N-17 Duplicated and Incorrect `BeaconChanged` Event Emission

In the `_setBeacon` function of the `BaseERC20` contract, the `BeaconChanged` event is emitted with the same address being used for the old and new value. Moreover, the same event is already emitted in the `upgradeBeaconToAndCall` function.

Consider removing the event emission in the `_setBeacon` function to only emit the event once, correctly.

Update: *Resolved in pull request #49 at commit 81afe2f and in pull request #51 at commit 9d02a5d.*

N-18 Assertions Not Failing Early

In the `_mint` function of the `BaseERC20` contract, the validation to protect against overflow in the total supply is performed after the `to` address' balance increment.

In order to fail early in the checks, follow the checks-effects-interactions (CEI) pattern, and to reduce the gas consumed during reverted mints, consider validating the overflow before the balance of the `to` address is incremented.

Update: *Resolved in pull request #52 at commit 1cee8fd.*

N-19 Misleading Clawback Functionality

The `ERC20RevocableStandard` contract introduces the `clawback` and `batchClawback` functions that enable the `REGISTRAR_ROLE` role to move funds at will without any restriction or prior authorization. However, the documentation for the clawback functionality states that the method "provides an allowance for an authorized party". This might seem misleading as it does not make use of any sort of allowance. In addition, since the authorized party is also in charge of other sensitive actions in the contracts (such as minting and burning tokens), it is not explicitly authorized by the owner of the assets to perform such an action.

Due to the importance and sensitive nature of the clawback functionality, consider rewriting the documentation to properly reflect the real intention of the functionality, along with descriptions of the proper situations in which it should be used, and the limitations that are placed on it. To go a step further, consider giving users the choice to to opt in if they want to enable the clawback feature for their balances.

Update: *Resolved in pull request #53 at commit 8ad711e. More documentation has been added.*

N-20 Implementation Contracts Are Left Uninitialized

Throughout the codebase, implementation contracts are left uninitialized. Instead, to prevent initialization by a malicious attacker, the `initialize` function relies on only allowing a specific address to call it. This is the case for the `initialize` functions in the `BaseERC20`, `ERC721BasicToken`, `ERC721SoulboundToken`, and `WhitelistComplianceOracle` contracts. While this method works, it does not follow the standard used by other proxy contracts whereby the `initialize` function is disabled in the constructor.

Consider following the standard and explicitly disabling the initializer in the constructors of the implementation contracts.

Update: *Resolved in pull request #54 at commit 6080b68.*

N-21 Misleading Comments

Throughout the codebase, multiple instances of misleading comments were identified:

- In the `WhitelistComplianceOracle` contract, it is checked if the implementation address has been set instead of validating whether the beacon can return the implementation contract.
- The docstring in the `ERC20BasicStandard` contract does not properly explain the purpose of this contract, which extends the functionality to include batch operations.
- In the `IERC20BatchClawback` interface, the `batchClawback` function is documented to emit the `TokensUnfrozen` event, which is not the case.

Consider addressing the above-listed instances of misleading comments to improve the clarity and maintainability of the codebase.

Update: *Resolved in pull request #42 at commit df9e43c.*

Conclusion

The audited codebase introduces a suite of ERC-20 smart contracts with upgradeable architecture, leveraging the beacon proxy pattern and integrating support for EIP-2612 permit functionality. The contracts grant specific administrative roles with the ability to manage token behavior, including upgrades and configuration of core parameters.

No critical or high-severity vulnerabilities were identified during the audit. Several lower-severity findings were noted, primarily relating to code clarity, documentation consistency, and test coverage. The development team was responsive and cooperative throughout the process, and promptly addressed all findings that warranted action.

Following the audit, the team enhanced the test suite to increase coverage and address previously missing edge case scenarios, contributing to the overall robustness of the system. While a few code structure patterns—such as repeated logic blocks and non-standard naming conventions—were observed, these do not impact the security of the contracts. Minor improvements in code organization and clarity are recommended to improve long-term maintainability.

From a security standpoint, the system is in a sound state, with well-documented functionality and no critical issues outstanding. Continued attention to testing, access control, and upgrade procedures will help ensure the integrity of the contracts over time.