Zellic

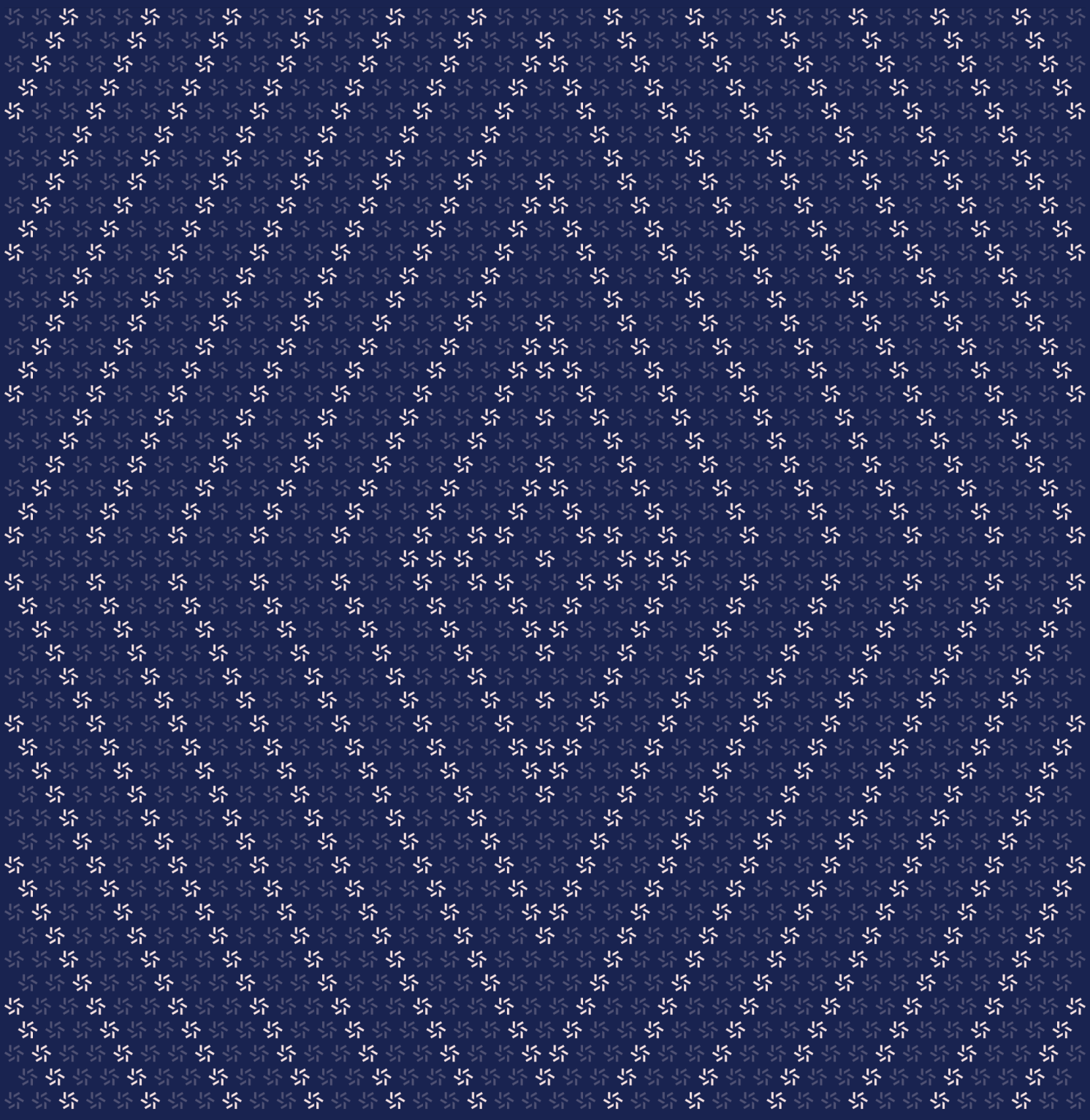
January 22, 2025

Prepared for
Dawson Reid
Joseph Onorati
Parket White
John Han
Apyx

Prepared by
Seunghyeon Kim
Kuilin Li
Zellic

Apyx Stablecoin

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction **6**

2.1. About Apyx Stablecoin 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Incorrect address accounting in CommitToken leads to permanent asset locking 11

3.2. Mishandled partial-vest accounting makes repeated withdrawals distribute unvested funds too early 13

3.3. Add controls to UnlockToken to limit deposits and redemption requests to the ApyUSD vault 15

3.4. Third-party–controlled distribution of vested assets may be unsafe 17

3.5. Nonce can be duplicated 19

3.6. The `validateOrder` view function has a misleading interface 21

3.7. The mint rate-limit implementation could be faster 23

3.8. Insufficient test coverage 25

4. Discussion 27

4.1. Potential inflation attack on ApyUSD 28

4.2. CommitTokens can be wrapped by third parties 29

5. Threat Model 31

5.1. Module: AddressList.sol 32

5.2. Module: ApxUSD.sol 35

5.3. Module: ApyUSD.sol 42

5.4. Module: CommitToken.sol 52

5.5. Module: ERC20FreezeableUpgradable.sol 69

5.6. Module: LinearVestV0.sol 72

5.7. Module: MinterV0.sol 78

5.8. Module: Roles.sol 89

5.9. Module: UnlockToken.sol 97

5.10. Module: YieldDistributor.sol 99

6. Assessment Results 102

6.1. Disclaimer 103

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.



1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Apyx from January 13th to January 16th, 2026. During this engagement, Zellic reviewed Apyx Stablecoin's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an on-chain attacker drain the Apyx locking vaults or extract unfair value by manipulating exchange rates via donation attacks or flash-loan exploits?
- Could a malicious message or signature replay trigger a permanent lock-up of user funds or allow unauthorized actors to bypass AccessManager restrictions?
- Is the codebase implemented to prevent operational griefing, such as forced cooldown resets, and does it maintain state consistency during complex ERC-7540 withdrawal flows?
- Could an attacker manipulate the yield distribution or the ApyUSD exchange rate by exploiting the LinearVestV0 vesting schedule or through arbitrary asset donations?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

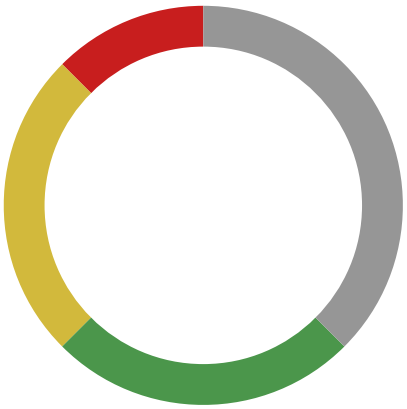
1.4. Results

During our assessment on the scoped Apyx Stablecoin contracts, we discovered eight findings. One critical issue was found. Two were of medium impact, two were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Apyx in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	0
Medium	2
Low	2
Informational	3



2. Introduction

2.1. About Apyx Stablecoin

Apyx contributed the following description of Apyx Stablecoin:

> Apyx is the first Dividend-Backed Stablecoin (DBS) protocol backed by variable rate perpetual preferred stock such as STRC issued by Strategy (MSTR).
>
> The protocol uses a two-token model:
>
> - **apxUSD**: Apyx's synthetic non-yield bearing stable asset intended to provide deep liquidity in secondary markets and broad usability as a collateral asset across DeFi and CeFi. apxUSD is intended to be over collateralized at all times.
> - **apyUSD**: Apyx's savings asset, a yield-bearing stablecoin (YBS) that accrues yield from dividends paid by Digital Asset Treasury (DAT) companies such as Strategy (MSTR).

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look

for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Apyx Stablecoin Contracts

Type	Solidity
Platform	EVM-compatible
Target	evm-contracts
Repository	https://github.com/apyx-labs/evm-contracts ↗
Version	`86d40fc3acdb24390875ed53299eede819b0cb61`

Programs	`AddressList`
	`ApxUSD`
	`ApyUSD`
	`CommitToken`
	`LinearVestV0`
	`MinterV0`
	`Roles`
	`UnlockToken`
	`YieldDistributor`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of eight person-days. The assessment was conducted by two consultants over the course of four calendar days.

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Seunghyeon Kim
Engineer
seunghyeon@zellic.io ↗

Kuilin Li
Engineer
kuilin@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 13, 2026	Kick-off call
January 13, 2026	Start of primary review period
January 16, 2026	End of primary review period

3. Detailed Findings

3.1. Incorrect address accounting in CommitToken leads to permanent asset locking

Target	CommitToken		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The CommitToken contract utilizes `msg.sender` as the key to track redemption requests in the `redeemRequests` mapping. When a user initiates a withdrawal via the ApyUSD vault, the vault calls `requestRedeem` on UnlockToken (a CommitToken instance). In this context, `msg.sender` is the ApyUSD contract address, not the user's address.

Consequently, all redemption requests (shares) are incorrectly aggregated under the single ApyUSD address instead of being tracked per individual user. In the validation logic `balanceOf(owner) - request.shares < shares`, `request.shares` reflects the accumulated total of all requests (attributed to the vault), while `balanceOf(owner)` reflects only the individual user's balance. This mismatch causes the check to fail for all subsequent users after the first one, triggering an `InsufficientBalance` error.

Impact

All subsequent users (after the first) are unable to withdraw funds via the ApyUSD vault. Since the withdrawal transaction reverts, users never receive UnlockToken shares. Without these shares, they cannot initiate a manual redemption request directly on the CommitToken as a workaround. This results in the permanent locking of user assets.

Recommendations

Modify the `requestRedeem` and `requestWithdraw` functions to use `owner` instead of `msg.sender` as the key for the `redeemRequests` mapping. This ensures that redemption requests are correctly tracked and isolated for each individual asset owner, even when the request is initiated by an operator like the vault.

Remediation

This issue has been acknowledged by Apyx, and a fix was implemented in commit a83f0676 ↗.

Apyx provided the following response:

Fixed the redeemRequests mapping key in CommitToken from msg.sender to owner across requestRedeem, requestWithdraw, and _withdraw, ensuring redemption requests are tracked per individual user rather than being incorrectly aggregated under the calling vault address. Three multi-user regression tests were added to verify that sequential withdrawals by different users through the ApyUSD vault no longer revert.

3.2. Mishandled partial-vest accounting makes repeated withdrawals distribute unvested funds too early

Target	LinearVestV0		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The LinearVestV0 contract allows a specific depositor to deposit funds that then vest to a specific beneficiary over time. During the configured vesting period, the total amount of unvested funds will linearly vest until all of it is vested at the end of the period.

The function that the beneficiary calls in order to withdraw the funds is `transferVestedYield`:

```
function transferVestedYield() external override onlyBeneficiary {
    uint256 vested = vestedAmount();
    vestingAmount -= vested;

    _transferVestedYield(vested);
}
```

The `_transferVestedYield` function does not update any internal accounting and only does the ERC-20 transfer of the underlying asset. The call to `vestedAmount` performs the following:

```
function vestedAmount() public view override returns (uint256) {
    // [...]
    uint256 timeSinceLastDeposit;
    unchecked {
        timeSinceLastDeposit = block.timestamp - lastDepositTimestamp;
    }
    // [...]
    return (vestingAmount * timeSinceLastDeposit) / vestingPeriod;
}
```

There is a logic error in this flow that leads to the unintended consequence where repeated calls to `transferVestedYield` will end up transferring nearly all of the unvested funds. For example, if exactly 1/3 of the vesting period has happened, then the quantity `timeSinceLastDeposit` divided by `vestingPeriod` will be 1/3. So the above logic will reduce `vestingAmount` by 2/3, transferring the 1/3 to the beneficiary. But no other state variables have changed here. So, if the beneficiary calls `transferVestedYield` again immediately, they will receive 1/3 of the 2/3 that was left from the

previous call.

As the number of calls becomes large, the amount of unvested funds that immediately become vested through this logic error approaches 100%.

Impact

Since the intended beneficiary of the LinearVest0 contract is the ApyUSD contract, which calls `$.vesting.transferVestedYield();` in its `_withdraw` function, this means that attackers who realize this behavior can easily engineer many small withdrawals from many accounts in order to realize unvested funds faster than the depositor intended.

This attack does not allow an attacker to access funds that belong to other users or break any invariants, but it does allow them to accelerate the vesting of the funds that would have gone to them.

Recommendations

We recommend reviewing the high-level design intent of the vesting process in order to amend this logic in an appropriate way.

Remediation

This issue has been acknowledged by Apyx, and a fix was implemented in commit `1e0b2d18` ↗.

Apyx provided the following response:

> Replaced the single-variable vesting model with dual-tracking accounting: a fullyVestedAmount accumulator for previously vested-but-unclaimed yield and a lastTransferTimestamp to calculate only the newly vested portion since the last withdrawal, preventing repeated calls to transferVestedYield (renamed to pullVestedYield in PR #66) from draining unvested funds. PR #66 further simplified the calculation by using Math.mulDiv with explicit period-remaining and period-elapsed windows, eliminating the original back-derivation of the starting vesting amount.

3.3. Add controls to UnlockToken to limit deposits and redemption requests to the ApyUSD vault

Target	CommitToken, UnlockToken		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

According to the documentation, UnlockToken functions are intended to be called exclusively by the ApyUSD contract. However, due to the lack of access control on functions like `deposit`, `requestRedeem`, and `requestWithdraw`, any user holding ApxUSD can directly deposit into and request redemptions from UnlockToken, bypassing the ApyUSD vault entirely.

Impact

This issue does not allow users to drain funds from the ApyUSD vault or bypass the withdrawal logic for funds already locked in the vault (users must still hold the underlying asset to use UnlockToken directly). However, it permits unintended interaction with a contract designed solely for vault usage. This theoretically allows the contract's state (e.g., total supply) to be affected by external actors, violating the intended system architecture and potentially complicating off-chain indexers or management scripts.

Recommendations

To align implementation with the design intent, it is recommended to restrict key functions (`deposit`, `requestRedeem`, etc.) in the UnlockToken contract using an `onlyVault` modifier, ensuring they can only be called by the immutable vault address.

Remediation

This issue has been acknowledged by Apyx, and a fix was implemented in commit `14027ebf` ↗.

Apyx provided the following response:

> Added an onlyVault modifier to UnlockToken that restricts _deposit and _requestRedeem to the immutable vault address, preventing external actors from directly interacting with UnlockToken and bypassing the ApyUSD vault. The parent CommitToken functions were

made virtual to enable the override, and a bug where redeem/withdraw returned stale values after request deletion was also fixed.

3.4. Third-party–controlled distribution of vested assets may be unsafe

Target	LinearVestV0		
Category	Protocol Risks	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The LinearVestV0 contract normally distributes vested funds to the beneficiary when the beneficiary calls the `transferVestedYield` function. However, in addition to that flow, it also distributes vested funds to the beneficiary in `depositYield` if there are any vested funds left:

```
function depositYield(uint256 amount) external override restricted {
    // [...]
    // Transfer out any vested yield before resetting (if any)
    _transferVestedYield(vested);
    // [...]
}
```

The `_transferVestedYield` function does an ERC-20 transfer of the vested funds to the beneficiary. The caller of the `depositYield` function is likely governance, not the beneficiary.

This means that if the beneficiary engages in any economic activity on other protocols or systems, then the depositor can potentially double spend assets by filling orders using this unexpected transfer.

For instance, if the beneficiary address has an open UniswapX order requesting the same asset, then the depositor can initiate the fill of that order and then call `depositYield` during the UniswapX fill callback. Since UniswapX will consider the depositor-initiated increase in the beneficiary's balance as satisfying the conditions of the limit order, it will allow the transaction to proceed.

Impact

The depositor can trick other systems into believing that it has caused an increase in balance in the beneficiary via this third-party distribution.

The impact of this is practically very low, since the depositor is likely the governance in this system, and both the depositor and the beneficiary are trusted to not mishandle assets in this way. However, if this LinearVestV0 contract is used in any other contexts, especially if the beneficiary address belongs to end users, then this should be documented and fixed.

Recommendations

We recommend remediating this issue alongside the remediation for Finding 3.2. ↗. If the amount of vested unwithdrawn funds is directly tracked, then this function can just increase that quantity to offset the decrease caused by its internal accounting changes. This will ensure that the funds go to where they should, while simultaneously ensuring that the beneficiary is not given any funds unexpectedly in a transaction it did not initiate.

Remediation

This issue has been acknowledged by Apyx, and a fix was implemented in commit `1e0b2d18` ↗.

Apyx provided the following response:

> Eliminated the inline _transferVestedYield call from depositYield and replaced it with an accounting update (fullyVestedAmount += vested), so vested funds are accumulated in storage rather than transferred during deposit. This ensures the beneficiary only receives tokens when it explicitly calls pullVestedYield, removing the risk of unexpected third-party–initiated transfers during deposit callbacks.

3.5. Nonce can be duplicated

Target	MinterV0		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The MinterV0 contract uses an EIP-712 signature scheme that includes a full `uint48` nonce to prevent replay attacks. However, when scheduling the mint operation via the AccessManager, the `_encodeOrderData` function appends only a truncated version of the nonce (`nonce % 256`) to the calldata:

```
function _encodeOrderData(Order memory order)
    internal view returns (bytes memory) {
    return abi.encodePacked(
        abi.encodeCall(apxUSD.mint, (order.beneficiary, order.amount)),
        bytes1(uint8(order.nonce % 256)) // Append nonce byte for uniqueness
    );
}
```

This truncation results in identical calldata generation for orders that share the same `beneficiary` and `amount` but have nonces differing by multiples of 256 (e.g., nonce `N` and `N + 256`). Since AccessManager derives the `operationId` from the calldata, these distinct orders map to the same `operationId`.

Impact

This collision creates a constraint on concurrent orders. If an order with nonce `N` is currently pending (scheduled but not yet executed) in the AccessManager, a user effectively cannot schedule another order with identical parameters and nonce `N + 256` until the first one is processed or canceled.

Attempting to request the second mint will revert because the AccessManager disallows scheduling an operation that is already pending. While this does not put funds at risk or allow replay attacks (due to the signature verification using the full nonce), it can temporarily block valid user actions if the operation queue is not actively managed.

Recommendations

To prevent `operationId` collisions entirely, it is recommended to include the full `order.nonce` (or a larger unique identifier) in the encoded data passed to AccessManager.

If the one-byte size constraint is intentional for gas optimization or other design reasons, ensure that off-chain infrastructure is robust enough to monitor pending orders and execute or cancel them promptly to avoid blocking future requests associated with the same modulo-256 slot.

Remediation

This issue has been acknowledged by Apyx, and fixes were implemented in the following commits:

- `c677f557` ↗
- `d8763631` ↗

Apyx provided the following response:

> Replaced the truncated nonce % 256 byte appended to the AccessManager calldata with the full nonce passed as a parameter to apxUSD.mint, ensuring every distinct order produces a unique operationId and eliminating collisions between orders whose nonces differ by multiples of 256.

3.6. The `validateOrder` view function has a misleading interface

Target	MinterV0		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `validateOrder` function is a view function that checks if a given order is valid:

```
/**
 * @notice Validates an order without executing it (useful for off-chain
    checks)
 * @param order The mint order to validate
 * @param signature The beneficiary's signature over the order
 * @return bool True if the order is valid
 */
// slither-disable-start timestamp
function validateOrder(Order calldata order, bytes calldata signature)
    public view returns (bool) {
    // Check time window validity
    if (order.notAfter < order.notBefore) {
        revert OrderInvalidTimeWindow();
    }

    // [...]
    return true;
}
```

According to the interface and documentation comments, this function returns true if the order is valid. The program logic actually never returns false — it either reverts if the order is invalid for any reason or returns true.

Although this behavior is technically correct, it is misleading. In the current in-scope code, there is only one caller of `validateOrder`, at the beginning of `requestMint`:

```
function requestMint(Order calldata order, bytes calldata signature)
    external
    // [...]
{
    // 1. Validate order (signature, nonce, expiry, amount)
```

```
    validateOrder(order, signature);

    // [...]
```

Note that the return value of `validateOrder` is not checked — it is relied upon to revert if the order is invalid, which is an implementation detail.

Impact

As of the time of writing, there is no impact, because `validateOrder` either returns true or reverts.

However, a future version of `validateOrder` may add logic that causes it to return false. In this case, `requestMint` will incorrectly disregard the return value.

Recommendations

We recommend removing the boolean return value and specifying that it will revert if the order is invalid.

Remediation

This issue has been acknowledged by Apyx, and a fix was implemented in commit `1bf8d2df` ↗.

Apyx provided the following response:

> Removed the misleading bool return value from validateOrder, changing its signature to a void function that reverts on invalid orders, so callers cannot silently ignore a future false return.

3.7. The mint rate-limit implementation could be faster

Target	MinterV0		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the MinterV0 contract, there is a time-based rate limit that is implemented using an OZ DoubleEndedQueue.

Each time a mint finishes, its amount is pushed onto the back of the queue, and then, when the rate limit needs to be checked for new mints, the `_cleanMintHistory` function is called in order to remove entries from the front of the queue until the oldest entry is within the time window. Then, the `rateLimitMinted` function is called, which iterates through the time window in order to determine the actual rate-limit quantity.

The rate-limit checking process runs in O(n) time and gas, where n is the number of mints in the time window. But, with some straightforward tweaks to this process, this can happen in constant time. The current cleanup process (as of the time of writing) also runs in linear time and gas, but with some more tweaks to that process, it could instead be logarithmic.

Impact

If many small mints are performed, then the gas costs for rate-limit checking and cleanup linearly increase with the number of mints, which is inefficient.

Recommendations

For the rate-limit checking process, instead of recording the amount minted in each entry, it could record the total cumulative amount minted from contract construction to this mint. If the start and end of the window is known, and these cumulative quantities are recorded instead, then subtracting the two will yield the amount minted in the window. This works in constant time, rather than iterating through all of the entries in the window.

One drawback of this design is that over a long period of minting and burning activity, the cumulative total amount minted may approach the integer limit used for it. If this is a concern, then we recommend explicitly storing the quantities modulo that limit, using modular subtraction to find the sum over the window. If the same integer limit applies to the rate limit, then this is safely guaranteed to find the actual sum, because the real sum cannot exceed the rate limit, which cannot exceed the modulus.

For the cleanup process, instead of removing entries one by one until the oldest entry is new enough to be in the window, it can instead conduct a binary search on the queue to find the largest timestamp that is outside the window. Then, it can directly move the queue's head pointer to point to that entry.

This process has some drawbacks. Firstly, although the OZ DoubleEndedQueue implementation could support clearing an arbitrary amount of entries from the start of the queue in constant time, its current interface does not support this. So the queue implementation must be extended or modified to add it. Furthermore, like the OZ `clear` function's documentation states, clearing those by moving pointers to invalidate them misses out on possible storage gas refunds from explicitly clearing the members. Due to these drawbacks, it is likely not worth the engineering effort to implement this algorithm.

Remediation

This issue has been acknowledged by Apyx, and a fix was implemented in commit `291241cf` ↗. This only changes the iteration order from the back to the front of the history queue.

Apyx provided the following response:

> Optimized rateLimitMinted() to iterate from the back (newest) to the front (oldest) of the mint history queue, enabling an early exit once records fall outside the rate limit window. This reduces gas costs from O(n) to O(k) where k is the number of recent mints within the current period, and introduces a MAX_RATE_LIMIT_PERIOD ceiling to ensure historical records are retained when the rate limit period is extended.

3.8. Insufficient test coverage

Target	Project-wide		
Category	Protocol Risks	**Severity**	Medium
Likelihood	N/A	**Impact**	Medium

Description

During the audit, we identified that the project's test suite lacks comprehensive coverage for several critical components and complex state transitions. While some happy-path scenarios are tested, there is a distinct lack of negative testing and integration scenarios that mimic real-world usage, particularly involving the interaction between the vault (ApyUSD) and its dependencies.

Specific examples of logic errors that were not caught by the existing test suite include the following.

1. **CommitToken aggregation bug.** The CommitToken contract utilizes `msg.sender` as the key to track redemption requests in the `redeemRequests` mapping. When a user initiates a withdrawal via the ApyUSD vault, the vault calls `requestRedeem` on UnlockToken (a CommitToken instance). In this context, `msg.sender` is the ApyUSD contract address, not the user's address. Consequently, all redemption requests are incorrectly aggregated under the single ApyUSD address. The validation logic `balanceOf(owner) - request.shares < shares` fails for subsequent users because `request.shares` reflects the accumulated total of all requests (attributed to the vault), while `balanceOf(owner)` reflects only the individual user's balance. This critical flaw renders the redemption mechanism unusable for multiple users.
2. **LinearVestV0 double counting.** In LinearVestV0, when `transferVestedYield` is called, it correctly transfers the vested amount and reduces `vestingAmount`. However, it does not reset `lastDepositTimestamp` or properly adjust the vesting schedule variables (`timeSinceLastDeposit`). If `transferVestedYield` is called repeatedly (e.g., when half the period has passed), it will calculate the vested fraction based on the original start time but apply it to the remaining (already reduced) amount. This leads to incorrect vesting calculations and potentially draining funds faster or slower than intended, which a robust property-based test would have caught.
3. **Lack of real-world integration scenarios.** The test suite (as of the time of writing) focuses heavily on isolated interactions. There is a significant lack of integration tests that simulate a live mainnet environment where multiple users interact with the protocol simultaneously (e.g., concurrent deposits, withdrawals, and vesting schedules). As seen with the CommitToken aggregation bug, these multiuser scenarios are critical for uncovering state management issues and logic flaws that isolated unit tests often miss.

When building a complex contract with multiple moving parts (state changes) and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios on every function that touches the contract's state.

Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts and all functions — not just surface-level functions. It is important to test the invariants required for ensuring security and also verify any mathematical properties from the project's specification. Additionally, testing cross-chain function calls and transfers is recommended to ensure the desired functionality.

Impact

Good test coverage has multiple effects:

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in the product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point may seem contradictory given the time investment to create and maintain tests. However, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks other code if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence.

Tests have your back here. They are an excellent indicator that the existing functionality was most likely not broken by a change to the code. Without a comprehensive test suite, the above benefits are lost. This increases the likelihood of bugs and vulnerabilities going unnoticed until after deployment, which can be costly and damaging to the project's reputation.

Recommendations

We recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Remediation

This issue has been acknowledged by Apyx, and a fix was implemented in commit `e783491a` ↗.

Apyx provided the following response:

Introduced a comprehensive Foundry invariant test suite (test/invariant/) with handler-based multi-actor architecture simulating concurrent deposits, withdrawals, minting, and yield distribution across the full protocol stack. The suite enforces 13 system-wide invariants — including apxUSD supply cap, apyUSD total asset accounting, unlock token 1:1 backing, vesting solvency, share price floor, and protocol-wide apxUSD conservation — using randomized actor sequences that would have caught the CommitToken aggregation bug (3.1) and the LinearVestV0 double-counting issue (3.2). Per-contract negative path and input validation tests were also expanded as detailed in Section 5 of the threat model.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Potential inflation attack on ApyUSD

The ApyUSD contract implements the ERC-4626 standard with an explicit override of `_decimalsOffset()` set to 0, as seen in the codebase.

```
/**
 * @notice Returns the decimals offset for inflation attack protection
 * @dev Can be overridden to add virtual shares/assets
 */
function _decimalsOffset() internal pure override returns (uint8) {
    return 0;
}
```

This configuration preserves a 1:1 decimal alignment between the vault and its underlying asset, which simplifies off-chain integrations and user experience. However, this design choice inherently foregoes the built-in virtual-reserve protection found in modern ERC-4626 implementations, leaving the vault susceptible to inflation attacks (also known as donation attacks) when it holds low liquidity.

In an inflation attack, a malicious actor can manipulate the exchange rate by depositing a negligible amount (e.g., 1 WEI) and subsequently donating a large sum of assets to the vault. This action artificially inflates the share price, causing subsequent legitimate deposits — if smaller than the inflated share price—to round down to zero shares, effectively leading to a loss of user funds. Even in the absence of a malicious actor, a high share price combined with low decimal precision can result in significant rounding losses for smaller deposits.

To address this risk while balancing integration requirements, we recommend evaluating two primary mitigation strategies. The first and robust option is to increase the `_decimalsOffset` to at least 9. This approach leverages the standard's virtual-share mechanism to make inflation attacks economically infeasible and improves internal precision for share calculations. However, it results in the vault reporting higher decimals (e.g., 27 decimals for an 18-decimal asset), which may necessitate updates to off-chain indexers and UIs.

Alternatively, if maintaining strict 18-decimal parity is a critical business requirement, the protocol must enforce a mandatory operational safeguard. Specifically, the deployer should mint a nontrivial amount of dead shares (e.g., $1,000 worth) to a burn address immediately upon deployment. This seeding process locks the initial exchange rate and ensures the vault's total supply never drops low enough to facilitate cheap manipulation, effectively securing the vault without altering its decimal configuration.

We have categorized this finding as a Discussion section item because the vulnerability's

exploitability relies on a specific mathematical edge case that is practically difficult to sustain against a rational market. The ApyUSD vault inherits from OpenZeppelin's ERC4626Upgradeable, utilizing the standard share conversion logic:

```
/**
 * @dev Internal conversion function (from assets to shares) with support for
    rounding direction.
 */
function _convertToShares(uint256 assets, Math.Rounding rounding)
    internal view virtual returns (uint256) {
    return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(),
    totalAssets() + 1, rounding);
}
```

With `_decimalsOffset()` set to 0, the formula simplifies to `shares = (assets * (totalSupply + 1)) / (totalAssets + 1)`. For an attacker to enforce a rounding-down-to-zero scenario (where `shares = 0`), the victim's deposited assets must be strictly less than the attacker's inflated `totalAssets` relative to the supply. Consequently, profitable exploitation would require a sequence of multiple victims (e.g., at least three) consecutively depositing amounts below this specific threshold to offset the attacker's initial donation cost. If even a single user deposits an amount slightly above the threshold, they would mint shares and immediately dilute the attacker's donated capital, causing a net loss for the attacker. This mandatory prerequisite — that zero shares must be minted repeatedly without interruption — makes the attack highly difficult. Any deposit exceeding the threshold invalidates the attacker's position, rendering the strategy economically irrational. Thus, we classify this as a Discussion point.

This issue has been acknowledged by Apyx, and a fix was implemented in commit d780fba3 ↗.

Apyx provided the following response:

> Introduced atomic deployer contracts (ApxUSDDeployer, ApyUSDDeployer) that deploy, initialize, and seed the vault with a minimum 10,000 apxUSD initial deposit in a single transaction, preventing both front-running of proxy initialization and inflation attacks by ensuring the vault's total supply is never low enough to facilitate exchange rate manipulation.

4.2. CommitTokens can be wrapped by third parties

One stated design goal of the CommitToken contract is that these lock tokens cannot be transferred. This is implemented via a restriction in the `_update` function override:

```
/**
```

```
 * @notice Lock tokens are not transferable and only support minting and
    burning
 * @inheritdoc ERC20
 */
function _update(address from, address to, uint256 value)
    internal override(ERC20, ERC20Pausable) {
    // Only support minting and burning
    if (from != address(0) && to != address(0)) {
        revert NotSupported();
    }
    super._update(from, to, value);
}
```

However, it is important to note that because of composability, it is not possible to completely restrict the transfer of lock tokens. Although the above override prevents ERC-20 transfers made on the contract itself, it does not prevent the owner from being a contract that delegates beneficial ownership to an address that can then be updated.

For instance, if a sophisticated trader wishes to unlock tokens, instead of directly unlocking them, they could deploy a wrapper contract that owns the lock tokens. This wrapper contract could then be freely traded, giving future ownership of the unlocked tokens to whoever owns it.

Due to the decentralized nature of blockchains, although more effective and complex methods can be employed to prevent trading assets, it is not functionally possible to absolutely prohibit it. For instance, it is possible to implement a dynamic whitelist where only addresses that have been proven to be EOAs (via the submission of any signature that recovers to the address) can hold the asset — but this mitigation can be sidestepped by deploying the wrapper contract into the EOA as an EIP-7702 contract, potentially with Nick's method to prove to any future recipient that the EOA's private key cannot sign anything else.

Even with a perfect on-chain implementation of nontransferability, off-chain implementations can still perform more-or-less effective wrapping in a similar way. For instance, two parties can sign a TradFi contract to transfer the future ownership of the locked tokens, or a holding company that holds the keys and on-chain ownership of the locked tokens could issue transferable shares of itself, or more.

Because of these considerations, if this transfer limitation is an implementation convenience rather than an economic design goal, then we recommend clarifying this in the documentation. Alternatively, assuming that this transfer limitation is not an economic goal, then we recommend adding functionality to mint lock tokens as NFTs so that unsophisticated traders can also get the liquidity benefit of being able to sell off the locked tokens before they finish unlocking.

This issue has been acknowledged by Apyx, and a fix was implemented in commit d51160e7 ↗.

Apyx provided the following response:

Added NatSpec documentation to CommitToken and UnlockToken clarifying that non-transferability is an implementation convenience for simplifying async redeem accounting, not an economic design invariant, and noting that future versions may support transferability or NFT-based unlocking positions.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

This threat model reflects a point-in-time snapshot of the codebase at commit 86d40fc3acdb24390875ed53299eede819b0cb61. Following this review, Apyx expanded the test suite; references to the additional tests are included beneath each section.

5.1. Module: AddressList.sol

The gap in test coverage was acknowledged by Apyx, and coverage was expanded in PR#59 ↗

Function: `add(address user)`

This function is for adding a new address to the address list. It is restricted to authorized roles via AccessManager.

Inputs

- `user`
  - **Control**: The caller (restricted).
  - **Constraints**: Must not be the zero address and must not already exist in the list.
  - **Impact**: Adds the address to the `_addresses` set and emits an `Added` event.

Branches and code coverage

Intended branches

- Successfully adds a new, valid address to the list.
  - ☐ Test coverage

Negative behavior

- Reverts if `user` is the zero address.
  - ☐ Negative test
- Reverts if `user` is already in the list.
  - ☐ Negative test

- Reverts if the caller is not authorized.
  - ☐ Negative test

Function: `at(uint256 index)`

This function is for returning the address stored at a specific index in the list.

Inputs

- `index`
  - **Control**: The caller.
  - **Constraints**: Must be less than the total number of addresses (length).
  - **Impact**: Selects which address is returned.

Branches and code coverage

Intended branches

- Returns the address at the valid index.
  - ☐ Test coverage

Negative behavior

- Reverts if `index` is out of bounds (implicit in EnumerableSet).
  - ☐ Negative test

Function: `constructor(address initialAuthority)`

This function is for initializing the AddressList contract with the specified AccessManager authority.

Inputs

- `initialAuthority`
  - **Control**: The deployer of the contract.
  - **Constraints**: Must not be the zero address.
  - **Impact**: Sets the `authority` for the AccessManaged contract, determining access control.

Branches and code coverage

Intended branches

- The contract is successfully initialized with a valid authority.
  - ☐ Test coverage

Negative behavior

- Reverts if `initialAuthority` is the zero address.
  - ☐ Negative test

Function: `contains(address user)`

This function is for checking if a given address is present in the address list.

Inputs

- `user`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Determines the boolean return value.

Branches and code coverage

Intended branches

- Returns `true` if the address is in the list.
  - ☐ Test coverage
- Returns `false` if the address is not in the list.
  - ☐ Test coverage

Function: `length()`

This function is for returning the total number of addresses in the list.

Branches and code coverage

Intended branches

- Returns the count of addresses.

- ☐ Test coverage

Function: `remove(address user)`

This function is for removing an address from the address list. It is restricted to authorized roles via AccessManager.

Inputs

- `user`
  - **Control**: The caller (restricted).
  - **Constraints**: Must exist in the list.
  - **Impact**: Removes the address from the `_addresses` set and emits a `Removed` event.

Branches and code coverage

Intended branches

- Successfully removes an existing address from the list.
  - ☐ Test coverage

Negative behavior

- Reverts if `user` is not in the list.
  - ☐ Negative test
- Reverts if the caller is not authorized.
  - ☐ Negative test

5.2. Module: ApxUSD.sol

The gap in test coverage was acknowledged by Apyx, and coverage was expanded in PR#63 ↗

Function: `_getApxUSDStorage()`

This function is for returning the storage pointer for the ApxUSD struct (ERC-7201).

Branches and code coverage

Intended branches

- Returns the storage pointer.
  - ☑ Test coverage

Function: `_update(address from, address to, uint256 value)`

This function is for the hook called before transfers. It calls super (pausable, freezeable).

Inputs

- `from`
  - **Control**: Internal.
  - **Constraints**: Not frozen.
  - **Impact**: Sender.
- `to`
  - **Control**: Internal.
  - **Constraints**: Not frozen.
  - **Impact**: Recipient.
- `value`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Amount.

Branches and code coverage

Intended branches

- Calls super.
  - ☑ Test coverage

Negative behavior

- Reverts if paused.
  - ☑ Negative test
- Reverts if frozen.
  - ☑ Negative test

Function: `constructor()`

This function is for disabling initializers for the implementation contract.

Branches and code coverage

Intended branches

- Calls `_disableInitializers`.
  - ☑ Test coverage

Function: `freeze(address target)`

This function is for freezing a target address. It is restricted to `ADMIN_ROLE`.

Inputs

- `target`
  - **Control**: Admin.
  - **Constraints**: None.
  - **Impact**: Account to freeze.

Branches and code coverage

Intended branches

- Freezes target.
  - ☑ Test coverage

Negative behavior

- Reverts if unauthorized.
  - ☑ Negative test

Function: `initialize(address initialAuthority, uint256 initialSupplyCap)`

This function is for initializing the proxy. Sets up ERC-20, Permit, Pausable, AccessManager, and supply cap.

Inputs

- `initialAuthority`
  - **Control**: Admin.
  - **Constraints**: Nonzero.
  - **Impact**: Access control.
- `initialSupplyCap`
  - **Control**: Admin.
  - **Constraints**: Positive.
  - **Impact**: Supply limit.

Branches and code coverage

Intended branches

- Initializes dependencies and storage.
  - ☑ Test coverage

Negative behavior

- Reverts if authority is zero.
  - ☐ Negative test
- Reverts if the cap is zero.
  - ☐ Negative test
- Reverts if already initialized.
  - ☐ Negative test

Function: `mint(address to, uint256 amount)`

This function is for minting new apxUSD tokens. It checks the supply cap. It is restricted to `MINT_STRAT_ROLE`.

Inputs

- `to`
  - **Control**: Admin.
  - **Constraints**: Not a banned address (checked in `_update`).
  - **Impact**: Token recipient.
- `amount`

- **Control**: Admin.
- **Constraints**: Must not exceed supply cap.
- **Impact**: Mint amount.

Branches and code coverage

Intended branches

- Checks supply cap and mints.
  - ☑ Test coverage

Negative behavior

- Reverts if supply cap is exceeded.
  - ☑ Negative test
- Reverts if paused (in `_update`).
  - ☑ Negative test
- Reverts if frozen (in `_update`).
  - ☑ Negative test

Function: `pause()`

This function is for pausing transfers. It is restricted to `ADMIN_ROLE`.

Branches and code coverage

Intended branches

- Pauses contract.
  - ☑ Test coverage

Negative behavior

- Reverts if unauthorized.
  - ☑ Negative test

Function: `setSupplyCap(uint256 newSupplyCap)`

This function is for updating the supply cap. It is restricted to `ADMIN_ROLE`.

Inputs

- `newSupplyCap`
  - **Control**: Admin.
  - **Constraints**: Cannot be less than the current total supply.
  - **Impact**: Supply limit.

Branches and code coverage

Intended branches

- Updates storage and emits an event.
  - ☑ Test coverage

Negative behavior

- Reverts if the new cap is less than the total supply.
  - ☑ Negative test

Function: `supplyCap()`

This function is for returning the current supply cap.

Branches and code coverage

Intended branches

- Returns the cap.
  - ☑ Test coverage

Function: `supplyCapRemaining()`

This function is for returning the remaining mintable amount.

Branches and code coverage

Intended branches

- Returns the cap minus the current supply.
  - ☑ Test coverage
- Returns `0` if the current supply is greater than or equal to the cap.

- ☑ Test coverage

Function: `unfreeze(address target)`

This function is for unfreezing a target address. It is restricted to `ADMIN_ROLE`.

Inputs

- `target`
  - **Control**: Admin.
  - **Constraints**: None.
  - **Impact**: Account to unfreeze.

Branches and code coverage

Intended branches

- Unfreezes the target.
  - ☑ Test coverage

Negative behavior

- Reverts if unauthorized.
  - ☑ Negative test

Function: `unpause()`

This function is for unpausing transfers. It is restricted to `ADMIN_ROLE`.

Branches and code coverage

Intended branches

- Unpauses the contract.
  - ☑ Test coverage

Negative behavior

- Reverts if unauthorized.
  - ☑ Negative test

5.3. Module: ApyUSD.sol

The gap in test coverage was acknowledged by Apyx, and coverage was expanded in commit 01d7059e7e5b65238700b4d90182ed2415f89537 ↗

Function: `_decimalsOffset()`

This function is for returning the decimals offset. It returns `0` for a 1:1 relationship with the asset.

Branches and code coverage

Intended branches

- Returns `0`.
  - ☑ Test coverage

Function: `_deposit(address caller, address receiver, uint256 assets, uint256 shares)`

This function is for internal depositing with denylist checking.

Inputs

- `caller`
  - **Control**: User.
  - **Constraints**: Not denied.
  - **Impact**: Depositor.
- `receiver`
  - **Control**: User.
  - **Constraints**: Not denied.
  - **Impact**: Shares recipient.
- `assets`
  - **Control**: User.
  - **Constraints**: None.
  - **Impact**: Deposit amount.
- `shares`
  - **Control**: User.
  - **Constraints**: None.

- **Impact**: Mint amount.

Branches and code coverage

Intended branches

- Checks denylist.
  - ☑ Test coverage
- Calls `super._deposit`.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller/receiver is denied.
  - ☐ Negative test

Function: `_getApyUSDStorage()`

This function returns a storage pointer for the ApyUSD struct (ERC-7201).

Branches and code coverage

Intended branches

- Returns the storage pointer.
  - ☐ Test coverage

Function: `_revertIfDenied(ApyUSDStorage $, address user)`

This function is for checking if a user is in the denyList.

Inputs

- `$`
  - **Control**: Internal.
  - **Constraints**: Storage pointer.
  - **Impact**: Access to denyList.
- `user`
  - **Control**: Internal.

- **Constraints**: None.
- **Impact**: User to check.

Branches and code coverage

Intended branches

- Checks denylist.
  - ☑ Test coverage

Negative behavior

- Reverts if the user is denied.
  - ☐ Negative test

Function: `_update(address from, address to, uint256 value)`

This function is for a hook called before transfers.

Inputs

- `from`
  - **Control**: Internal.
  - **Constraints**: Not frozen.
  - **Impact**: Sender.
- `to`
  - **Control**: Internal.
  - **Constraints**: Not frozen.
  - **Impact**: Recipient.
- `value`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Amount.

Branches and code coverage

Intended branches

- Calls super.

- ☑ Test coverage

Negative behavior

- Reverts if paused (in super).
  - ☐ Negative test
- Reverts if frozen (in super).
  - ☐ Negative test

Function: `_withdraw(address caller, address receiver, address owner, uint256 assets, uint256 shares)`

This function is for executing a withdrawal by moving assets to UnlockToken and requesting redemption. It checks user compliance, cleans up vested yield, burns shares, and delegates real withdrawal to UnlockToken.

Inputs

- `caller`
  - **Control**: User.
  - **Constraints**: Not denied.
  - **Impact**: Initiator.
- `receiver`
  - **Control**: User.
  - **Constraints**: Not denied.
  - **Impact**: Recipient of the UnlockToken position.
- `owner`
  - **Control**: User.
  - **Constraints**: Not denied.
  - **Impact**: Share owner.
- `assets`
  - **Control**: User.
  - **Constraints**: None.
  - **Impact**: Amount to withdraw.
- `shares`
  - **Control**: User.
  - **Constraints**: None.
  - **Impact**: Amount to burn.

Branches and code coverage

Intended branches

- Validation (denyList, `unlockToken` set).
  - ☐ Test coverage
- Vested yield claim.
  - ☐ Test coverage
- Share burning (`super._withdraw` inside structure).
  - ☐ Test coverage
- Deposits to `unlockToken`.
  - ☐ Test coverage
- Requests `Redeem` on `unlockToken`.
  - ☐ Test coverage

Negative behavior

- Reverts if users denied.
  - ☐ Negative test
- Reverts if unlockToken is not set.
  - ☐ Negative test
- Reverts if unlockToken deposit fails.
  - ☐ Negative test

Function: `constructor()`

This function disables the initializers for the implementation contract.

Branches and code coverage

Intended branches

- Calls `_disableInitializers`.
  - ☐ Test coverage

Function: `decimals()`

This function is for returning token decimals. It inherits from ERC-4626 (matches asset).

Branches and code coverage

Intended branches

- Returns decimals.
  - ☑ Test coverage

Function: `denyList()`

This function is for returning the current DenyList contract address.

Branches and code coverage

Intended branches

- Returns address.
  - ☑ Test coverage

Function: `freeze(address target)`

This function is for freezing a target address. It is restricted to `ADMIN_ROLE`.

Inputs

- `target`
  - **Control**: Admin.
  - **Constraints**: None.
  - **Impact**: Account to freeze.

Branches and code coverage

Intended branches

- Freezes target.
  - ☐ Test coverage

Function: `initialize(address initialAuthority, address asset, address initialDenyList)`

This function is for initializing the vault. It sets up ERC-20, Permit, Pausable, ERC-4626, AccessManager, and DenyList.

Inputs

- `initialAuthority`
  - **Control**: Admin.
  - **Constraints**: Nonzero.
  - **Impact**: Access control.
- `asset`
  - **Control**: Admin.
  - **Constraints**: Nonzero.
  - **Impact**: Underlying asset (ApxUSD).
- `initialDenyList`
  - **Control**: Admin.
  - **Constraints**: Nonzero.
  - **Impact**: Compliance check.

Branches and code coverage

Intended branches

- Initializes a valid contract.
  - ☑ Test coverage

Negative behavior

- Reverts if any input is zero.
  - ☑ Negative test
- Reverts if already initialized.
  - ☑ Negative test

Function: `pause()`

This function is for pausing the contract. It is restricted to `ADMIN_ROLE`.

Branches and code coverage

Intended branches

- Pauses the contract.
  - ☐ Test coverage

Function: `setDenyList(IAddressList newDenyList)`

This function is for setting the DenyList contract address. It is restricted to `ADMIN_ROLE`.

Inputs

- `newDenyList`
  - **Control**: Admin.
  - **Constraints**: Nonzero.
  - **Impact**: Compliance check.

Branches and code coverage

Intended branches

- Updates storage and emits an event.
  - ☐ Test coverage

Negative behavior

- Reverts if the address is zero.
  - ☐ Negative test

Function: `setUnlockToken(IUnlockToken newUnlockToken)`

This function is for setting the UnlockToken contract address. It is restricted to `ADMIN_ROLE`.

Inputs

- `newUnlockToken`
  - **Control**: Admin.
  - **Constraints**: Nonzero.
  - **Impact**: Contract for withdrawals.

Branches and code coverage

Intended branches

- Updates storage and emits an event.
  - ☐ Test coverage

Negative behavior

- Reverts if the address is zero.
  - ☐ Negative test

Function: `setVesting(IVesting newVesting)`

This function is for setting the Vesting contract address. It is restricted to `ADMIN_ROLE`.

Inputs

- `newVesting`
  - **Control**: Admin.
  - **Constraints**: None (checks are usage-based).
  - **Impact**: Contract for yield.

Branches and code coverage

Intended branches

- Updates storage and emits an event.
  - ☐ Test coverage

Function: `totalAssets()`

This function is for returning the total assets, including those held in the vault and vested in the vesting contract.

Branches and code coverage

Intended branches

- Returns balance plus vested.
  - ☑ Test coverage

Function: `unfreeze(address target)`

This function is for unfreezing a target address. It is restricted to `ADMIN_ROLE`.

Inputs

- `target`
  - **Control**: Admin.
  - **Constraints**: None.
  - **Impact**: Account to unfreeze.

Branches and code coverage

Intended branches

- Unfreezes target.
  - ☐ Test coverage

Function: `unlockToken()`

This function is for returning the current UnlockToken address.

Branches and code coverage

Intended branches

- Returns the address.
  - ☐ Test coverage

Function: `unpause()`

This function is for unpausing the contract. It is restricted to `ADMIN_ROLE`.

Branches and code coverage

Intended branches

- Unpauses the contract.
  - ☐ Test coverage

Function: `vesting()`

This function is for returning the current Vesting address.

Branches and code coverage

Intended branches

- Returns the address.
  - ☐ Test coverage

5.4. Module: CommitToken.sol

The gap in test coverage was acknowledged by Apyx, and coverage was expanded in commit a72559eb7a61f3239c7e2cb06aa39e0a47437948 ↗

Function: `_convertToAssets(uint256 shares, Math.Rounding)`

This function is for converting shares to assets (1:1).

Inputs

- `shares`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Conversion base.
- `<unnamed@1>`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Rounding (ignored).

Branches and code coverage

Intended branches

- Returns equal amount.
  - ☑ Test coverage

Function: `_convertToShares(uint256 assets, Math.Rounding)`

This function is for converting assets to shares (1:1).

Inputs

- `assets`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Conversion base.
- `<unnamed@1>`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Rounding (ignored).

Branches and code coverage

Intended branches

- Returns equal amount.
  - ☑ Test coverage

Function: `_cooldownRemaining(Request request)`

This function is for checking the cooldown (internal helper).

Inputs

- `request`
  - **Control**: Internal (storage).
  - **Constraints**: None.
  - **Impact**: Returns remaining time.

Branches and code coverage

Intended branches

- Returns `0` if the request is not set or time has passed.
  - ☑ Test coverage

- Returns remaining seconds otherwise.

  ☑ Test coverage

Function: `_deposit(address caller, address receiver, uint256 assets, uint256 shares)`

This function is for overriding checks to include the denylist before deposit.

Inputs

- `caller`
  - **Control**: Internal.
  - **Constraints**: Must not be denied.
  - **Impact**: Depositor.
- `receiver`
  - **Control**: Internal.
  - **Constraints**: Must not be denied.
  - **Impact**: Share recipient.
- `assets`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Amount.
- `shares`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Share amount.

Branches and code coverage

Intended branches

- Checks denylist and calls super.

  ☑ Test coverage

Negative behavior

- Reverts if paused.

  ☐ Negative test
- Reverts if the caller/receiver denied.

- ☐ Negative test

Function: `_isClaimable(Request request)`

This function is for checking claimability (internal helper).

Inputs

- `request`
  - **Control**: Storage.
  - **Constraints**: None.
  - **Impact**: Boolean status.

Branches and code coverage

Intended branches

- Returns true if request exists and cooldown is `0`.
  - ☑ Test coverage

Function: `_requestRedeem(Request request, address controller, address owner, uint256 assets, uint256 shares)`

This function is for processing redeem/withdraw requests (internal helper). It updates the request state for the user.

Inputs

- `request`
  - **Control**: Internal (storage pointer).
  - **Constraints**: None.
  - **Impact**: User's request state.
- `controller`
  - **Control**: The caller.
  - **Constraints**: Must be the operator of the owner and operator of `msg.sender`.
  - **Impact**: Who controls the request.
- `owner`
  - **Control**: The caller.

- **Constraints**: Must own shares.
- **Impact**: Who owns shares.

- `assets`
  - **Control**: Calculated.
  - **Constraints**: None.
  - **Impact**: Asset amount in request.
- `shares`
  - **Control**: The caller.
  - **Constraints**: Must have balance.
  - **Impact**: Share amount to lock.

Branches and code coverage

Intended branches

- Updates request and emits an event.
  - ☑ Test coverage

Negative behavior

- Reverts if unauthorized (operator check).
  - ☑ Negative test
- Reverts if there is insufficient balance.
  - ☑ Negative test
- Reverts if denied.
  - ☐ Negative test

Function: `_revertIfDenied(address user)`

This function is for checking the denylist (internal helper).

Inputs

- `user`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Checks if the address is blocked.

Branches and code coverage

Intended branches

- Does nothing if the user is not denied.
  - ☐ Test coverage

Negative behavior

- Reverts if the user is denied.
  - ☐ Negative test

Function: `_update(address from, address to, uint256 value)`

This function is for updating token balances. It enforces nontransferability (only mint/burn).

Inputs

- `from`
  - **Control**: Internal.
  - **Constraints**: Zero for mint.
  - **Impact**: Sender.
- `to`
  - **Control**: Internal.
  - **Constraints**: Zero for burn.
  - **Impact**: Receiver.
- `value`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Amount.

Branches and code coverage

Intended branches

- Mints tokens (`from=0`).
  - ☑ Test coverage
- Burns tokens (`to=0`).
  - ☑ Test coverage

Negative behavior

- Reverts if transfer attempted (`from!=0 && to!=0`).
  - ☑ Negative test

Function: `_withdraw(Request request, address caller, address receiver, address owner)`

This function is for processing withdrawals/redeems (claims) — internal helper.

Inputs

- `request`
  - **Control**: Internal (storage).
  - **Constraints**: Must exist and be claimable.
  - **Impact**: Request to claim.
- `caller`
  - **Control**: Internal.
  - **Constraints**: Must be `msg.sender`.
  - **Impact**: Claimant.
- `receiver`
  - **Control**: Internal.
  - **Constraints**: Must not be denied.
  - **Impact**: Recipient.
- `owner`
  - **Control**: Internal.
  - **Constraints**: Must be `msg.sender`. Not denied.
  - **Impact**: Share owner.

Branches and code coverage

Intended branches

- Deletes request and calls super `_withdraw`.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller/owner `!= msg.sender`.

☑ Negative test

- Reverts if denied.

☐ Negative test

- Reverts if not claimable.

☑ Negative test

Function: `claimableRedeemRequest(uint256, address owner)`

This function is for returning claimable shares.

Inputs

- `<unnamed@0>`
  - **Control**: Ignored.
  - **Constraints**: None.
  - **Impact**: None.
- `owner`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Query target.

Branches and code coverage

Intended branches

- Returns `0` if not requested or not claimable.

☑ Test coverage

- Returns shares if claimable.

☑ Test coverage

Function: `constructor(address authority_, address asset_, uint48 unlockingDelay_, address denyList_)`

This function is for initializing the CommitToken contract.

Inputs

- `authority_`

- **Control**: The deployer.
- **Constraints**: Must not be zero address.
- **Impact**: AccessManager for control.

- `asset_`
  - **Control**: The deployer.
  - **Constraints**: Must not be zero address.
  - **Impact**: Underlying asset (IERC-20).
- `unlockingDelay_`
  - **Control**: The deployer.
  - **Constraints**: Must not be zero.
  - **Impact**: Cooldown period for redemptions.
- `denyList_`
  - **Control**: The deployer.
  - **Constraints**: Must not be zero address.
  - **Impact**: AddressList for denials.

Branches and code coverage

Intended branches

- Successfully initializes the contract.
  - ☑ Test coverage

Negative behavior

- Reverts if inputs are invalid.
  - ☑ Negative test

Function: `cooldownRemaining(uint256, address owner)`

This function is for checking the cooldown (public view).

Inputs

- `<unnamed@0>`
  - **Control**: Ignored.
  - **Constraints**: None.
  - **Impact**: None.
- `owner`

- **Control**: The caller.
- **Constraints**: None.
- **Impact**: Query target.

Branches and code coverage

Intended branches

- Returns the result of `_cooldownRemaining`.

☑ Test coverage

Function: `decimals()`

This function is for returning asset decimals.

Branches and code coverage

Intended branches

- Returns decimals.

☐ Test coverage

Function: `isClaimable(uint256, address owner)`

This function is for checking claimability (public view).

Inputs

- `<unnamed@0>`
  - **Control**: Ignored.
  - **Constraints**: None.
  - **Impact**: None.
- `owner`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Query target.

Branches and code coverage

Intended branches

- Returns the result of `_isClaimable`.
  - ☑ Test coverage

Function: `isOperator(address controller, address operator)`

This function is for checking if an operator is authorized for a controller.

Inputs

- `controller`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Account being controlled.
- `operator`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Account to check authorization for.

Branches and code coverage

Intended branches

- Returns true if the operator equals the controller.
  - ☐ Test coverage

Function: `maxRedeem(address owner)`

This function is for returning the max shares valid for redemption (claimable).

Inputs

- `owner`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Query target.

Branches and code coverage

Intended branches

- Returns claimable amount.
  - ☑ Test coverage

Function: `pause()`

This function is for pausing the contract. It is restricted.

Branches and code coverage

Intended branches

- Pauses the contract.
  - ☐ Test coverage

Negative behavior

- Reverts if unauthorized.
  - ☐ Negative test

Function: `pendingRedeemRequest(uint256, address owner)`

This function is for returning pending shares (not yet claimable).

Inputs

- `<unnamed@0>`
  - **Control**: Ignored.
  - **Constraints**: None.
  - **Impact**: None.
- `owner`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Query target.

Branches and code coverage

Intended branches

- Returns 0 if claimable.
  - ☑ Test coverage
- Returns shares if pending.
  - ☑ Test coverage

Function: `redeem(uint256 shares, address receiver, address owner)`

This function is for claiming a pending redeem request.

Inputs

- `shares`
  - **Control**: The caller.
  - **Constraints**: Must match the requested shares.
  - **Impact**: Amount to claim.
- `receiver`
  - **Control**: The caller.
  - **Constraints**: Not denied.
  - **Impact**: Asset recipient.
- `owner`
  - **Control**: The caller.
  - **Constraints**: Must be `msg.sender`.
  - **Impact**: Share owner.

Branches and code coverage

Intended branches

- Calls `_withdraw` and returns assets.
  - ☑ Test coverage

Negative behavior

- Reverts if shares mismatch.
  - ☑ Negative test
- Reverts if `_withdraw` checks fail.

☑ Negative test

Function: `requestRedeem(uint256 shares, address controller, address owner)`

This function is for initiating a redeem request. It locks the potential rate.

Inputs

- `shares`
  - **Control**: The caller.
  - **Constraints**: Must have balance.
  - **Impact**: Amount to redeem.
- `controller`
  - **Control**: The caller.
  - **Constraints**: Must be the operator.
  - **Impact**: Controller address.
- `owner`
  - **Control**: The caller.
  - **Constraints**: Must own shares.
  - **Impact**: Owner address.

Branches and code coverage

Intended branches

- Calculates assets and calls `_requestRedeem`.

☑ Test coverage

Negative behavior

- Reverts if checks in `_requestRedeem` fail.

☑ Negative test

Function: `requestWithdraw(uint256 assets, address controller, address owner)`

This function is for initiating a withdraw request. It locks the potential rate.

Inputs

- `assets`
  - **Control**: The caller.
  - **Constraints**: Must imply valid shares.
  - **Impact**: Amount to withdraw.
- `controller`
  - **Control**: The caller.
  - **Constraints**: Must be the operator.
  - **Impact**: Controller address.
- `owner`
  - **Control**: The caller.
  - **Constraints**: Must own shares.
  - **Impact**: Owner address.

Branches and code coverage

Intended branches

- Calculates shares and calls `_requestRedeem`.
  - ☑ Test coverage

Negative behavior

- Reverts if checks in `_requestRedeem` fail.
  - ☑ Negative test

Function: `setDenyList(address newDenyList)`

This function is for setting the DenyList contract. It is restricted to the admin.

Inputs

- `newDenyList`
  - **Control**: The admin.
  - **Constraints**: Must not be zero address.
  - **Impact**: Updates blocking logic.

Branches and code coverage

Intended branches

- Updates denylist and emits an event.
  - ☐ Test coverage

Negative behavior

- Reverts if zero address.
  - ☐ Negative test
- Reverts if unauthorized.
  - ☐ Negative test

Function: `setOperator(address, bool)`

This function is not implemented.

Inputs

- `<unnamed@0>`
  - **Control**: Ignored.
  - **Constraints**: None.
  - **Impact**: None.
- `<unnamed@1>`
  - **Control**: Ignored.
  - **Constraints**: None.
  - **Impact**: None.

Branches and code coverage

Intended branches

- None.

Negative behavior

- Always reverts with `NotSupported`.
  - ☐ Negative test

Function: `setUnlockingDelay(uint48 newUnlockingDelay)`

This function is for setting the new unlocking delay. It is restricted to the admin.

Inputs

- `newUnlockingDelay`
  - **Control**: The admin.
  - **Constraints**: None specific (`uint48`).
  - **Impact**: Changes cooldown for future requests.

Branches and code coverage

Intended branches

- Updates delay and emits an event.
  - ☐ Test coverage

Negative behavior

- Reverts if unauthorized.
  - ☐ Negative test

Function: `unpause()`

This function is for unpausing the contract. It is restricted.

Branches and code coverage

Intended branches

- Unpauses the contract.
  - ☐ Test coverage

Negative behavior

- Reverts if unauthorized.
  - ☐ Negative test

Function: `withdraw(uint256 assets, address receiver, address owner)`

This function is for claiming a pending withdraw request.

Inputs

- `assets`
  - **Control**: The caller.
  - **Constraints**: Must match requested assets.
  - **Impact**: Amount to claim.
- `receiver`
  - **Control**: The caller.
  - **Constraints**: Not denied.
  - **Impact**: Asset recipient.
- `owner`
  - **Control**: The caller.
  - **Constraints**: Must be `msg.sender`.
  - **Impact**: Share owner.

Branches and code coverage

Intended branches

- Calls `_withdraw` and returns shares.
  - ☑ Test coverage

Negative behavior

- Reverts if assets mismatch.
  - ☑ Negative test
- Reverts if `_withdraw` checks fail.
  - ☑ Negative test

5.5. Module: ERC20FreezeableUpgradable.sol

Function: `_freeze(address target)`

This function is for freezing a target address. It prevents the address from sending or receiving tokens.

Inputs

- `target`
  - **Control**: Internal.
  - **Constraints**: Nonzero.
  - **Impact**: Account to freeze.

Branches and code coverage

Intended branches

- Sets frozen status to true and emits an event.
  - ☑ Test coverage

Negative behavior

- Reverts if the target is zero address.
  - ☑ Negative test

Function: `_getFreezeableStorage()`

This function is for returning the storage pointer for the `FreezeableStorage` struct (ERC-7201).

Branches and code coverage

Intended branches

- Returns the storage pointer.
  - ☑ Test coverage

Function: `_unfreeze(address target)`

This function is for unfreezing a target address. It allows the address to send or receive tokens again.

Inputs

- `target`
  - **Control**: Internal.
  - **Constraints**: None.

- **Impact**: Account to unfreeze.

Branches and code coverage

Intended branches

- Removes frozen status and emits an event.
  - ☑ Test coverage

Function: `_update(address from, address to, uint256 value)`

This function is for a hook that is called before any token transfer. It enforces that neither the sender nor the recipient is frozen.

Inputs

- `from`
  - **Control**: Internal.
  - **Constraints**: Not frozen.
  - **Impact**: Sender.
- `to`
  - **Control**: Internal.
  - **Constraints**: Not frozen.
  - **Impact**: Recipient.
- `value`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Amount.

Branches and code coverage

Intended branches

- Calls `super._update` if checks pass.
  - ☑ Test coverage

Negative behavior

- Reverts if `from` is frozen.
  - ☑ Negative test

- Reverts if `to` is frozen.
  - ☑ Negative test

Function: `isFrozen(address target)`

This function is for checking if a target address is frozen.

Inputs

- `target`
  - **Control**: User.
  - **Constraints**: None.
  - **Impact**: Account to check.

Branches and code coverage

Intended branches

- Returns the freeze status of the target.
  - ☑ Test coverage

5.6. Module: LinearVestV0.sol

Function: `_transferVestedYield(uint256 _vestedAmount)`

This function is for executing the transfer of vested tokens (internal helper).

Inputs

- `_vestedAmount`
  - **Control**: The caller (internal).
  - **Constraints**: None (if 0, returns early).
  - **Impact**: Amount to transfer.

Branches and code coverage

Intended branches

- Returns early if the amount is 0.

☑ Test coverage

- Transfers tokens to beneficiary and emits an event.

☑ Test coverage

Negative behavior

- Reverts if the transfer fails.

☑ Negative test

Function: `_unvestedAmount(uint256 _vestedAmount)`

This function is for calculating the unvested amount given the vested amount (internal helper).

Inputs

- `_vestedAmount`
  - **Control**: The caller (internal).
  - **Constraints**: Should be `<= vestingAmount`.
  - **Impact**: Used to calculate the difference.

Branches and code coverage

Intended branches

- Returns the difference between the total vesting amount and vested amount.

☑ Test coverage

Function: `constructor(address _asset, address _authority, address _beneficiary, uint256 _vestingPeriod)`

This function is for initializing the LinearVestV0 contract.

Inputs

- `_asset`
  - **Control**: The deployer.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The token to be vested.
- `_authority`

- **Control**: The deployer.
- **Constraints**: Must not be the zero address.
- **Impact**: The AccessManager for access control.

- `_beneficiary`
  - **Control**: The deployer.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The contract authorized to receive vested tokens.
- `_vestingPeriod`
  - **Control**: The deployer.
  - **Constraints**: Must not be zero.
  - **Impact**: The duration over which yield vests.

Branches and code coverage

Intended branches

- Successfully initializes with valid parameters.
  - ☑ Test coverage

Negative behavior

- Reverts if any address is zero.
  - ☑ Negative test
- Reverts if the vesting period is zero.
  - ☑ Negative test

Function: `depositYield(uint256 amount)`

This function is for depositing new yield to be vested. This function transfers any currently vested yield (but not claimed) to the beneficiary, adds the new amount to the unvested portion, and resets the vesting schedule.

Inputs

- `amount`
  - **Control**: The caller.
  - **Constraints**: Must be greater than zero.
  - **Impact**: Increases the total amount vesting.

Branches and code coverage

Intended branches

- Calculates current vested amount and unvested amount.
  - ☑ Test coverage
- Resets vesting period with new total (unvested plus amount).
  - ☑ Test coverage
- Transfers previously vested amount to beneficiary.
  - ☑ Test coverage
- Transfers `amount` from the caller to this contract.
  - ☑ Test coverage

Negative behavior

- Reverts if `amount` is 0.
  - ☑ Negative test
- Reverts if transfer fails.
  - ☑ Negative test

Function: `setBeneficiary(address newBeneficiary)`

This function is for updating the beneficiary address. It is restricted to `ADMIN_ROLE`.

Inputs

- `newBeneficiary`
  - **Control**: The caller (admin).
  - **Constraints**: Must not be zero address.
  - **Impact**: New recipient of vested tokens.

Branches and code coverage

Intended branches

- Updates beneficiary and emits an event.
  - ☑ Test coverage

Negative behavior

- Reverts if `newBeneficiary` is zero address.
  - ☑ Negative test
- Reverts if the caller is unauthorized.
  - ☑ Negative test

Function: `setVestingPeriod(uint256 newPeriod)`

This function is for updating the vesting period. It is restricted to `ADMIN_ROLE`.

Inputs

- `newPeriod`
  - **Control**: The caller (admin).
  - **Constraints**: Must not be zero.
  - **Impact**: Affects vesting schedule of future deposits and unvested amounts.

Branches and code coverage

Intended branches

- Updates vesting period and emits an event.
  - ☑ Test coverage

Negative behavior

- Reverts if `newPeriod` is 0.
  - ☑ Negative test
- Reverts if the caller is unauthorized.
  - ☑ Negative test

Function: `transferVestedYield()`

This function is for transferring available vested yield to the beneficiary. It is restricted to calls by the beneficiary (Silo).

Branches and code coverage

Intended branches

- Transfers vested amount if nonzero.
  - ☑ Test coverage
- Updates vesting state.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the beneficiary.
  - ☑ Negative test

Function: `unvestedAmount()`

This function is for calculating the amount of yield that is currently unvested.

Branches and code coverage

Intended branches

- Returns `vestingAmount` minus `vestedAmount`.
  - ☑ Test coverage

Function: `vestedAmount()`

This function is for calculating the amount of yield that has vested based on linear vesting from the last deposit time.

Branches and code coverage

Intended branches

- Returns `0` if vesting amount is zero.
  - ☑ Test coverage
- Returns total vesting amount if the vesting period has passed.
  - ☑ Test coverage
- Returns linearly calculated amount if partially vested.
  - ☑ Test coverage

5.7. Module: MinterV0.sol

The gap in test coverage was acknowledged by Apyx, and coverage was expanded in commit a72559eb7a61f3239c7e2cb06aa39e0a47437948 ↗

Function: `_cleanMintHistory()`

This function is for cleaning expired mint records from the queue (internal helper).

Branches and code coverage

Intended branches

- Calls `_cleanMintHistoryUpTo`.
  - ☑ Test coverage

Function: `_cleanMintHistoryUpTo(uint32 n)`

This function is for cleaning up to `n` expired mint records from the queue (internal helper).

Inputs

- `n`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Limit cleanup.

Branches and code coverage

Intended branches

- Iterates queue and removes expired records.
  - ☑ Test coverage
- Stops at cutoff or `n`.
  - ☑ Test coverage

Function: `_decodeMintRecord(bytes32 data)`

This function is for decoding `bytes32` into a mint record.

Inputs

- `data`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Encoded record.

Branches and code coverage

Intended branches

- Returns the decoded struct.
  - ☑ Test coverage

Function: `_encodeMintRecord(uint48 timestamp, uint208 amount)`

This function is for encoding a mint record into `bytes32`.

Inputs

- `timestamp`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Record time.
- `amount`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Record amount.

Branches and code coverage

Intended branches

- Returns packed `bytes32`.
  - ☑ Test coverage

Function: `_encodeOrderData(Order order)`

This function is for encoding order data for AccessManager scheduling/execution.

Inputs

- `order`
  - **Control**: Internal.
  - **Constraints**: None.
  - **Impact**: Encoded data.

Branches and code coverage

Intended branches

- Returns encoded data with nonce byte.
  - ☑ Test coverage

Function: `cancelMint(bytes32 operationId)`

This function is for canceling a pending mint operation. It cleans storage and cancels in AccessManager.

Inputs

- `operationId`
  - **Control**: The caller.
  - **Constraints**: Must exist in `pendingOrders`.
  - **Impact**: Operation to cancel.

Branches and code coverage

Intended branches

- Verifies existence.
  - ☑ Test coverage
- Deletes from storage (CEI).
  - ☑ Test coverage
- Calls `manager.cancel`.
  - ☑ Test coverage

Negative behavior

- Reverts if order is not found.

☑ Negative test

Function: `cleanMintHistory(uint32 n)`

This function is for manually cleaning up to `n` expired mint records from the history queue.

Inputs

- `n`
  - **Control**: The caller.
  - **Constraints**: Higher values cost more gas.
  - **Impact**: Cleanup count.

Branches and code coverage

Intended branches

- Cleans up to `n` records.

☑ Test coverage

Negative behavior

- Reverts if unauthorized.

☑ Negative test

Function: `constructor(address initialAuthority, address _apxUSD, uint208 _maxMintAmount, uint208 _rateLimitAmount, uint48 _rateLimitPeriod)`

This function is for initializing the MinterV0 contract with the EIP-712 domain separator and initial configuration.

Inputs

- `initialAuthority`
  - **Control**: Deployer.
  - **Constraints**: Must not be zero address.
  - **Impact**: AccessManager for role control.
- `_apxUSD`

- **Control**: Deployer.
- **Constraints**: Must not be zero address.
- **Impact**: The ApxUSD token contract.

- `_maxMintAmount`
  - **Control**: Deployer.
  - **Constraints**: Must not be zero.
  - **Impact**: Maximum mint per order.
- `_rateLimitAmount`
  - **Control**: Deployer.
  - **Constraints**: Must not be zero.
  - **Impact**: Maximum mint per period.
- `_rateLimitPeriod`
  - **Control**: Deployer.
  - **Constraints**: Must not be zero.
  - **Impact**: Rate-limit duration.

Branches and code coverage

Intended branches

- Successfully initializes the contract.
  - ☑ Test coverage

Negative behavior

- Reverts if any input is zero (address or amount).
  - ☐ Negative test

Function: `executeMint(bytes32 operationId)`

This function is for executing a previously scheduled mint operation. It verifies order validity and executes via AccessManager.

Inputs

- `operationId`
  - **Control**: The caller.
  - **Constraints**: Must exist in `pendingOrders`.
  - **Impact**: Operation to execute.

Branches and code coverage

Intended branches

- Checks order expiry and existence.
  - ☑ Test coverage
- Deletes order from storage (CEI).
  - ☑ Test coverage
- Calls `manager.execute`.
  - ☑ Test coverage

Negative behavior

- Reverts if order is not found.
  - ☑ Negative test
- Reverts if order expired.
  - ☑ Negative test
- Reverts if manager execution fails.
  - ☑ Negative test

Function: `hashOrder(Order order)`

This function is for computing the EIP-712 structural hash of the mint order.

Inputs

- `order`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Order to hash.

Branches and code coverage

Intended branches

- Returns a valid hash.
  - ☑ Test coverage

Function: `mintStatus(bytes32 operationId)`

This function is for returning the current status of a mint operation (`NotFound`, `Scheduled`, `Ready`, `Expired`).

Inputs

- `operationId`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Query ID.

Branches and code coverage

Intended branches

- Returns `NotFound` if not in storage.
  - ☑ Test coverage
- Returns `Expired` if past `notAfter`.
  - ☑ Test coverage
- Returns `Scheduled` if AccessManager delay has not passed.
  - ☑ Test coverage
- Returns `Ready` otherwise.
  - ☑ Test coverage

Function: `pause()`

This function is for pausing the minting process.

Branches and code coverage

Intended branches

- Pauses the contract.
  - ☑ Test coverage

Negative behavior

- Reverts if unauthorized.
  - ☑ Negative test

Function: `pendingOrder(bytes32 operationId)`

This function is for returning the details of a pending order.

Inputs

- `operationId`
  - **Control**: The caller.
  - **Constraints**: None.
  - **Impact**: Query ID.

Branches and code coverage

Intended branches

- Returns the order struct.
  - ☑ Test coverage

Function: `rateLimit()`

This function is for returning the current rate-limit configuration.

Branches and code coverage

Intended branches

- Returns the amount and period.
  - ☑ Test coverage

Function: `rateLimitAvailable()`

This function is for returning the amount available to mint without exceeding the rate limit.

Branches and code coverage

Intended branches

- Calculates the remaining capacity.
  - ☑ Test coverage

Function: `rateLimitMinted()`

This function is for returning the total amount minted in the current rate-limit period.

Branches and code coverage

Intended branches

- Calculates the total minted in window.
  - ☑ Test coverage

Function: `requestMint(Order order, bytes signature)`

This function is for requesting a mint operation. It validates order, checks rate limits, and schedules the operation via AccessManager.

Inputs

- `order`
  - **Control**: The caller.
  - **Constraints**: Must be a valid signed order.
  - **Impact**: Mint parameters.
- `signature`
  - **Control**: The caller.
  - **Constraints**: Must be valid.
  - **Impact**: Authorization.

Branches and code coverage

Intended branches

- Usage of `validateOrder` to check the signature.
  - ☑ Test coverage
- Usage of rate limits.
  - ☑ Test coverage
- Usage of AccessManager `schedule`.
  - ☑ Test coverage

Negative behavior

- Reverts if validation fails.
  - ☑ Negative test
- Reverts if rate limit is exceeded.
  - ☑ Negative test

Negative behavior

- Reverts if validation fails.
  - ☑ Negative test
- Reverts if rate limit is exceeded.
  - ☑ Negative test

Function: `setMaxMintAmount(uint208 newMaxMintAmount)`

This function is for updating the maximum mint amount per order.

Inputs

- `newMaxMintAmount`
  - **Control**: The caller.
  - **Constraints**: Must be positive.
  - **Impact**: New limit.

Branches and code coverage

Intended branches

- Updates storage and emits an event.
  - ☑ Test coverage

Negative behavior

- Reverts if zero.
  - ☐ Negative test

Function: `setRateLimit(uint256 newAmount, uint48 newPeriod)`

This function is for updating the rate-limit configuration. It cleans up old history.

Inputs

- `newAmount`
  - **Control**: The caller.
  - **Constraints**: Must be positive.
  - **Impact**: New rate-limit amount.
- `newPeriod`
  - **Control**: The caller.
  - **Constraints**: Must be positive.
  - **Impact**: New rate-limit period.

Branches and code coverage

Intended branches

- Updates storage, cleans history, and emits an event.
  - ☑ Test coverage

Negative behavior

- Reverts if the amount or period is zero.
  - ☐ Negative test

Function: `unpause()`

This function is for unpausing the minting process.

Branches and code coverage

Intended branches

- Unpauses the contract.
  - ☑ Test coverage

Negative behavior

- Reverts if unauthorized.
  - ☑ Negative test

Function: `validateOrder(Order order, bytes signature)`

This function is for validating a mint order structure and signature. It checks time windows, nonce, amount limits, and signature validity.

Inputs

- `order`
  - **Control**: The caller.
  - **Constraints**: Valid timestamps, nonce, and amount.
  - **Impact**: Order checking.
- `signature`
  - **Control**: The caller.
  - **Constraints**: Valid signature from beneficiary.
  - **Impact**: Signature verification.

Branches and code coverage

Intended branches

- Returns true if all checks pass.
  - ☑ Test coverage

Negative behavior

- Reverts if timestamps are invalid.
  - ☑ Negative test
- Reverts if the nonce is incorrect.
  - ☑ Negative test
- Reverts if the amount is greater than the maximum.
  - ☑ Negative test
- Reverts if the signature is invalid.
  - ☑ Negative test

5.8. Module: Roles.sol

Function: `assignAdminTargetsFor(AccessManager self, ApxUSD apxUSD)`

This function is for assigning administrative function selectors for the ApxUSD contract to the `ADMIN_ROLE`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `apxUSD`
  - **Control**: The caller.
  - **Constraints**: Valid ApxUSD contract.
  - **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `ADMIN_ROLE` permission for pause, unpause, `setSupplyCap`, freeze, and unfreeze.
  - ☐ Test coverage

Function: `assignAdminTargetsFor(AccessManager self, ApyUSD apyUSD)`

This function is for assigning administrative function selectors for the ApyUSD contract to the `ADMIN_ROLE`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `apyUSD`
  - **Control**: The caller.
  - **Constraints**: Valid ApyUSD contract.
  - **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `ADMIN_ROLE` permission for pause, unpause, freeze, unfreeze, `setDenyList`, `setUnlockToken`, and `setVesting`.

- ☐ Test coverage

Function: `assignAdminTargetsFor(AccessManager self, IAddressList denyList)`

This function is for assigning administrative function selectors for the AddressList (`denyList`) contract to the `ADMIN_ROLE`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `denyList`
  - **Control**: The caller.
  - **Constraints**: Valid AddressList contract.
  - **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `ADMIN_ROLE` permission for add and remove.
  - ☐ Test coverage

Function: `assignAdminTargetsFor(AccessManager self, IMinterV0 minterContract)`

This function is for assigning administrative function selectors for the MinterV0 contract to the `ADMIN_ROLE`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `minterContract`

- **Control**: The caller.
- **Constraints**: Valid Minter contract.
- **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `ADMIN_ROLE` permission for `setMaxMintAmount` and `setRateLimit`.
  - ☐ Test coverage

Function: `assignAdminTargetsFor(AccessManager self, IVesting vestingContract)`

This function is for assigning administrative function selectors for the Vesting contract to the `ADMIN_ROLE`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `vestingContract`
  - **Control**: The caller.
  - **Constraints**: Valid Vesting contract.
  - **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `ADMIN_ROLE` permission for `setVestingPeriod` and `setBeneficiary`.
  - ☐ Test coverage

Function: `assignAdminTargetsFor(AccessManager self, IYieldDistributor yieldDistributor)`

This function is for assigning administrative function selectors for the YieldDistributor contract to the `ADMIN_ROLE`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `yieldDistributor`
  - **Control**: The caller.
  - **Constraints**: Valid YieldDistributor contract.
  - **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `ADMIN_ROLE` permission for `setVesting`.
  - ☐ Test coverage

Function: `assignMintGuardTargetsFor(AccessManager self, IMinterV0 minterContract)`

This function is for assigning guardian function selectors for the MinterV0 contract to the `MINT_GUARD_ROLE`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `minterContract`
  - **Control**: The caller.
  - **Constraints**: Valid Minter contract.

- **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `MINT_GUARD_ROLE` permission for `cancelMint`.
  - ☐ Test coverage

Function: `assignMinterTargetsFor(AccessManager self, IMinterV0 minterContract)`

This function is for assigning minter function selectors for the MinterV0 contract to the `MINTER_ROLE`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `minterContract`
  - **Control**: The caller.
  - **Constraints**: Valid Minter contract.
  - **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `MINTER_ROLE` permission for `requestMint`, `executeMint`, and `cleanMintHistory`.
  - ☐ Test coverage

Function: `assignMintingContractTargetsFor(AccessManager self, ApxUSD apxUSD)`

This function is for assigning minting function selectors for the ApxUSD contract to the `MINT_STRAT_ROLE`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `apxUSD`
  - **Control**: The caller.
  - **Constraints**: Valid ApxUSD contract.
  - **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `MINT_STRAT_ROLE` permission for mint.
  - ☐ Test coverage

Negative behavior

- None (internal extension function).

Function: `assignYieldDistributorTargetsFor(AccessManager self, IVesting vestingContract)`

This function is for assigning yield-distribution function selectors for the Vesting contract to the `YIELD_DISTRIBUTOR_ROLE`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `vestingContract`
  - **Control**: The caller.
  - **Constraints**: Valid Vesting contract.
  - **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `YIELD_DISTRIBUTOR_ROLE` permission for `depositYield`.
  - ☐ Test coverage

Function: `assignYieldOperatorTargetsFor(AccessManager self, IYieldDistributor yieldDistributor)`

This function is for assigning yield-operator function selectors for the YieldDistributor contract to the `ROLE_YIELD_OPERATOR`.

Inputs

- `self`
  - **Control**: The caller.
  - **Constraints**: Valid AccessManager.
  - **Impact**: Grants permissions.
- `yieldDistributor`
  - **Control**: The caller.
  - **Constraints**: Valid YieldDistributor contract.
  - **Impact**: Target for permissions.

Branches and code coverage

Intended branches

- Grants `ROLE_YIELD_OPERATOR` permission for `depositYield`.
  - ☐ Test `coverage`

Function: `setRoleAdmins(AccessManager self)`

This function is for setting the admin role for various system roles to `ADMIN_ROLE`. This ensures centralized control.

Inputs

- `self`

- **Control**: The caller (must be authorized to call this extension).
- **Constraints**: Must be a valid AccessManager instance.
- **Impact**: Configures the role administration hierarchy in the AccessManager.

Branches and code coverage

Intended branches

- Successfully sets `ADMIN_ROLE` as admin for all defined system roles.
  - ☐ Test coverage

Negative behavior

- Reverts if called by an unauthorized user (depending on consumption).
  - ☐ Negative test

5.9. Module: UnlockToken.sol

The gap in test coverage was acknowledged by Apyx, and coverage was expanded in commit a72559eb7a61f3239c7e2cb06aa39e0a47437948 ↗

Function: `constructor(address authority_, address asset_, address vault_, uint48 unlockingDelay_, address denyList_)`

This function is for initializing the UnlockToken contract, setting up the CommitToken base and defining the vault address.

Inputs

- `authority_`
  - **Control**: The deployer.
  - **Constraints**: Valid AccessManager address.
  - **Impact**: Sets access control.
- `asset_`
  - **Control**: The deployer.
  - **Constraints**: Valid ERC-20 token address.
  - **Impact**: Underlying asset for the token.
- `vault_`
  - **Control**: The deployer.

- **Constraints**: Must not be zero address.
- **Impact**: Address authorized to act as operator for all users.

- `unlockingDelay_`
  - **Control**: The deployer.
  - **Constraints**: Valid duration.
  - **Impact**: Delay for redeeming tokens.
- `denyList_`
  - **Control**: The deployer.
  - **Constraints**: Valid AddressList address.
  - **Impact**: Address list for blocking functionality.

Branches and code coverage

Intended branches

- Successfully initializes the contract with valid parameters.
  - ☐ Test coverage

Negative behavior

- Reverts if `vault_` is the zero address.
  - ☐ Negative test

Function: `isOperator(address controller, address operator)`

This function is for checking if an operator is authorized for a given controller (token holder). The vault is always an authorized operator.

Inputs

- `controller`
  - **Control**: The caller or system checking permissions.
  - **Constraints**: None.
  - **Impact**: The address owning the tokens.
- `operator`
  - **Control**: The caller or system checking permissions.
  - **Constraints**: None.
  - **Impact**: The address attempting to operate.

Branches and code coverage

Intended branches

- Returns `true` if `operator` is the `controller`.
  - ☐ Test coverage
- Returns `true` if `operator` is the `vault`.
  - ☐ Test coverage
- Returns `false` otherwise.
  - ☐ Test coverage

Function: `name()`

This function is for returning the name of the token, ensuring it includes the "Unlock Token" suffix.

Branches and code coverage

Intended branches

- Returns "{AssetName} Unlock Token".
  - ☐ Test coverage

Function: `symbol()`

This function is for returning the symbol of the token, ensuring it includes the "_unlock" suffix.

Branches and code coverage

Intended branches

- Returns "{AssetSymbol}_unlock".
  - ☐ Test coverage

5.10. Module: YieldDistributor.sol

The gap in test coverage was acknowledged by Apyx, and coverage was expanded in commit a72559eb7a61f3239c7e2cb06aa39e0a47437948 ↗

Function: `availableBalance()`

This function is for returning the current balance of asset tokens held by this contract that can be distributed as yield.

Branches and code coverage

Intended branches

- Returns the balance of the asset token held by this contract.

☑ Test coverage

Function: `constructor(address _asset, address _authority, address _vesting)`

This function is for initializing the YieldDistributor contract with essential dependencies: the asset token, the access authority, and the vesting contract.

Inputs

- `_asset`
  - **Control**: The deployer.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The token to be distributed (apxUSD).
- `_authority`
  - **Control**: The deployer.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The AccessManager contract for access control.
- `_vesting`
  - **Control**: The deployer.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The destination contract for yield deposits.

Branches and code coverage

Intended branches

- Successfully initializes the contract with valid addresses.

☑ Test coverage

Negative behavior

- Reverts if `_asset` is the zero address.
  - ☑ Negative test
- Reverts if `_authority` is the zero address.
  - ☑ Negative test
- Reverts if `_vesting` is the zero address.
  - ☑ Negative test

Function: `depositYield(uint256 amount)`

This function is for transferring a specified amount of yield from this contract to the vesting contract. This function handles the approval process and triggers the deposit on the vesting contract. It is restricted to `YIELD_DISTRIBUTOR_ROLE`.

Inputs

- `amount`
  - **Control**: The caller (operator).
  - **Constraints**: Must be greater than zero and less than or equal to the available balance.
  - **Impact**: Determining the amount of tokens transferred to vesting.

Branches and code coverage

Intended branches

- Successfully approves and deposits yield to the vesting contract.
  - ☑ Test coverage

Negative behavior

- Reverts if the vesting contract is not set (zero address).
  - ☐ Negative test
- Reverts if `amount` is 0.
  - ☑ Negative test
- Reverts if the contract balance is insufficient.
  - ☑ Negative test
- Reverts if the caller is not authorized.

☑ Negative test

Function: `setVesting(address newVesting)`

This function is for updating the vesting contract address. This operation is restricted to the `ADMIN_ROLE`.

Inputs

- `newVesting`
  - **Control**: The caller (admin).
  - **Constraints**: Must not be the zero address.
  - **Impact**: Changes the destination for future yield deposits.

Branches and code coverage

Intended branches

- Successfully updates the vesting contract address and emits an event.
  ☑ Test coverage

Negative behavior

- Reverts if `newVesting` is the zero address.
  ☑ Negative test
- Reverts if the caller is not authorized (AccessManager).
  ☑ Negative test

6. Assessment Results

During our assessment on the scoped Apyx Stablecoin contracts, we discovered eight findings. One critical issue was found. Two were of medium impact, two were of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.