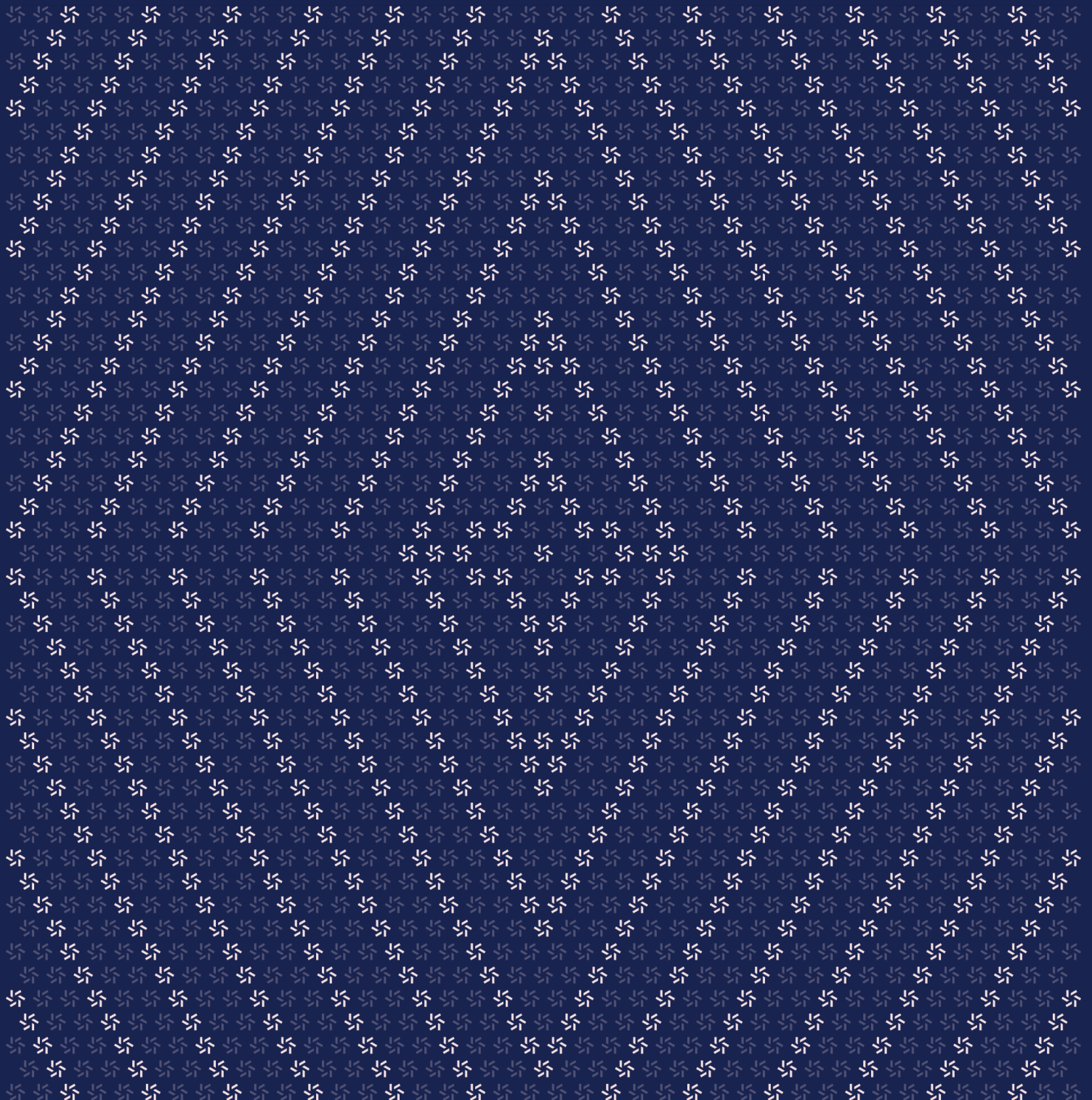


August 26, 2025

Axis

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 Axis	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. StakedAxisUSD can be permanently bricked through direct transfers	11
3.2. Insolvency for AxisUSDMinting	13
3.3. Users may be unable to withdraw from StakedAxisUSD due to minimum-shares check	15
3.4. A blacklisted user can still unstake from StakedAxisUSDV2	17
3.5. Fully restricted addresses can still deposit	19
3.6. Per-block mint/redeem caps enable same-block denial of service via executor ordering	21
3.7. Lack of comprehensive tests	23

3.8.	No method to withdraw Ether in AxisUSDMinting	25
3.9.	Missing check to prevent silo from being blacklisted	26
3.10.	Admin could get blacklisted through transferAdmin or grantRole	27
4.	Discussion	27
4.1.	Front-running concerns for StakedAxisUSDV2	28
4.2.	Custody wallet configuration and redemption liquidity	28
5.	Threat Model	28
5.1.	Module: AxisUSDMinting.sol	29
5.2.	Module: StakedAxisUSDV2.sol	38
5.3.	Module: StakedAxisUSD.sol	42
6.	Assessment Results	47
6.1.	Disclaimer	48

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 AxisFDN from August 18th to August 22nd, 2025. During this engagement, Zellic reviewed Axis'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:

- Are there any common issues with ERC-4626?
 - Are there cryptographic mistakes related to signed messages?
 - Does on-chain reward distribution work properly?
-

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.

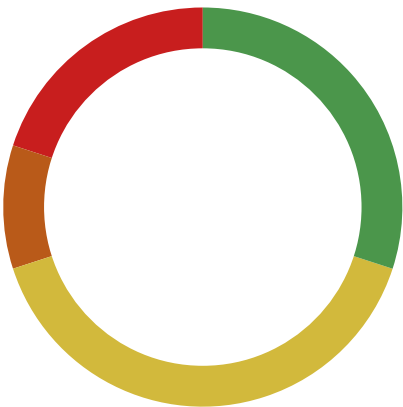
1.4. Results

During our assessment on the scoped Axis contracts, we discovered 10 findings. Two critical issues were found. One was of high impact, four were of medium impact, and three were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of AxisFDN in the Discussion section ([4.7](#)).

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	1
Medium	4
Low	3
Informational	0



2. Introduction

2.1. About Axis

AxisFDN contributed the following description of Axis:

Axis is a synthetic asset issuance and staking protocol that enables users to mint synthetic assets and participate in staking rewards distribution.

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. 7) 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:

Axis Contracts

Type	Solidity
Platform	EVM-compatible
Target	contracts
Repository	https://github.com/Axis-Coordinate/contracts ↗
Version	845d7651abfb4d114b40602adc1cffde97cac004
Programs	src/*.sol

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of two person-weeks. The assessment was conducted by three consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
✈ Engagement Manager
jacob@zellic.io ↗

Chad McDonald
✈ Engagement Manager
chad@zellic.io ↗

Pedro Moura
✈ Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Juchang Lee
✈ Engineer
lee@zellic.io ↗

Quentin Lemauf
✈ Engineer
quentin@zellic.io ↗

Rainier Wu
✈ Engineer
rainier@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

August 18, 2025 Kick-off call

August 18, 2025 Start of primary review period

August 22, 2025 End of primary review period

3. Detailed Findings

3.1. StakedAxisUSD can be permanently bricked through direct transfers

Target	StakedAxisUSD		
Category	Business Logic	Severity	Critical
Likelihood	High	Impact	Critical

Description

StakedAxisUSD implements a check for the minimum amount of shares if totalSupply is greater than zero, to prevent vault donation/inflation attacks.

```

/// @notice ensures a small non-zero amount of shares does not remain, exposing
to donation attack
function _checkMinShares() internal view {
    uint256 _totalSupply = totalSupply();
    if (_totalSupply > 0 && _totalSupply < MIN_SHARES) {
        revert MinSharesViolation();
    }
}

```

However, this allows for the following scenario:

1. StakedAxisUSD is deployed. Initially, the price of assets to shares will be 1:1.
2. An attacker directly transfers 1 Axis USD to StakedAxisUSD. Since vault assets have increased, the price of assets to shares is now 1e18:1, as Axis USD has 18 decimals.
3. A user intends to deposit. Due to _checkMinShares, any user who wishes to deposit must mint at least MIN_SHARES amount of shares, which is set to 1 ether = 1e18. As one share costs 1 Axis USD and a minimum of 1e18 shares have to be minted, this requires 1e18 Axis USD, which is infeasible to obtain.

This results in the inability for any users to use the vault, as no one can deposit.

Impact

The first deposit requires an infeasible amount of Axis USD, preventing users from depositing. Since the admin cannot resolve this denial of service through removing the asset tokens by calling rescueTokens, and since asset tokens cannot be rescued, this results in the vault becoming permanently unusable.

Recommendations

For protection against inflation/donation attacks, enable the ERC-4626 virtual-shares mechanism by overriding `_decimalsOffset()` to a nonzero value (e.g., 6–12).

The latest version of the ERC-4626 OpenZeppelin [contract](#) `z` explains their proposed solution to this type of attack:

The `_decimalsOffset()` corresponds to an offset in the decimal representation between the underlying asset's decimals and the vault decimals. This offset also determines the rate of virtual shares to virtual assets in the vault, which itself determines the initial exchange rate.

While not fully preventing the attack, analysis shows that the default offset (0) makes it non-profitable, as a result of the value being captured by the virtual shares (out of the attacker's donation) matching the attacker's expected gains. With a larger offset, the attack becomes orders of magnitude more expensive than it is profitable.

Remediation

This issue has been acknowledged by AxisFDN, and a fix was implemented in commit [f14abf93](#) `z`.

3.2. Insolvency for AxisUSDMinting

Target	AxisUSDMinting		
Category	Business Logic	Severity	Critical
Likelihood	High	Impact	Critical

Description

There are several ways that AxisUSDMinting could become insolvent, that is, unable to pay out funds through redeem.

The contract could become insolvent when price changes for the collateral token against Axis USD. The protocol intends to use stablecoins such as USDT and USDe; however, even with stablecoins, price may fluctuate slightly. Consider a case where USDT to Axis USD is 1:1. Alice mints 10 Axis USD, costing her 10 USDT. Then, the price of USDT rises slightly, and USDT to Axis USD is 1.01:1. Now, Alice wants to redeem 10 Axis USD for 10.1 USDT, but the contract only has 10 USDT and is unable to fulfill the redemption.

Another case in which the contract can become insolvent is when different tokens are involved. Consider if USDT and USDe were both whitelisted collateral tokens in AxisUSDMinting. Alice first uses USDT for collateral, minting 10 Axis USD, costing her 10 USDT. Then she redeems, choosing to receive 10 USDe for 10 Axis USD. However, the contract does not have any USDe and is unable to fulfill the redemption.

Impact

There are multiple ways in which AxisUSDMinting could become insolvent, and the likelihood of this occurring is relatively high. Insolvency would pose a serious risk, as it would prevent users from withdrawing their assets and ultimately harm both the users and the protocol.

Recommendations

Ensure AxisUSDMinting does not become insolvent. This may include the protocol transferring collateral tokens directly to AxisUSDMinting and performing rebalancing actions to maintain solvency. Because off-chain components and operational procedures are outside the scope of this audit, we cannot confirm whether measures are already in place to mitigate this risk.

These actions taken to prevent insolvency should be well-documented for users to maintain trust in the protocol.

Remediation

AxisFDN has acknowledged this issue and provided the following statement:

No code changes needed. Off-chain pricing handles cross-asset rates; manual rebalancing maintains reserves. Risk is operational, not smart contract-related.

3.3. Users may be unable to withdraw from StakedAxisUSD due to minimum-shares check

Target	StakedAxisUSD		
Category	Business Logic	Severity	High
Likelihood	High	Impact	High

Description

StakedAxisUSD implements a check for the minimum amount of shares if `totalSupply` is greater than zero, to prevent vault donation/inflation attacks — `MIN_SHARES` is set to 1 ether.

```

/// @notice ensures a small non-zero amount of shares does not remain, exposing
to donation attack
function _checkMinShares() internal view {
    uint256 _totalSupply = totalSupply();
    if (_totalSupply > 0 && _totalSupply < MIN_SHARES) {
        revert MinSharesViolation();
    }
}

```

However, consider the following scenario:

1. Alice deposits assets and mints `MIN_SHARES` (1 Ether) of shares.
2. Bob deposits assets and mints `MIN_SHARES` (1 Ether) of shares.
3. Alice redeems `MIN_SHARES - 1` (1e18 - 1 Ether) of shares.
4. Bob tries to redeem `MIN_SHARES` (1 Ether) of shares but fails, as this would result in one share left in the vault. Bob can only redeem one share, leaving the vault to contain 1 of Alice's shares and 1e18 - 1 of Bob's shares.

Impact

If every user were to withdraw everything from the vault, every user would need to redeem all of their shares. If any user does not redeem all of their shares, the last user may be unable to withdraw their shares.

An attacker can easily cause this scenario by intentionally withdrawing all but 1 of their shares. This leads to the last user being unable to redeem their shares for assets, causing loss of funds.

Recommendations

By design, the current guard enforces this behavior. Either

1. formally accept and document the constraint (including operational/user impacts), or
2. remove the guard to allow a user to withdraw all of their shares.

Please note that if the ERC-4626 virtual-share defense is adopted (see Finding [3.1](#), ↗), this guard provides limited value and can be safely removed.

Remediation

This issue has been acknowledged by AxisFDN, and a fix was implemented in commit [f14abf93](#) ↗.

3.4. A blacklisted user can still unstake from StakedAxisUSDV2

Target	StakedAxisUSDV2		
Category	Business Logic	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

StakedAxisUSD implements two blacklist roles, with two restriction levels.

```

/// @notice The role which prevents an address to stake
bytes32 private constant SOFT_RESTRICTED_STAKER_ROLE
    = keccak256("SOFT_RESTRICTED_STAKER_ROLE");
/// @notice The role which prevents an address to transfer, stake, or unstake.
    The owner of the contract can redirect address staking balance if an
    address is in full restricting mode.
bytes32 private constant FULL_RESTRICTED_STAKER_ROLE
    = keccak256("FULL_RESTRICTED_STAKER_ROLE");

```

If a user is fully restricted, they should not be able to unstake from the vault.

However, if the cooldown period is set and a user starts cooldown to claim assets by calling either `cooldownAssets` or `cooldownShares`, then the user becomes blacklisted with full restrictions, and they can still call `unstake` to claim their assets.

This is because during cooldown, assets are initially transferred to the silo, then after the cooldown period has ended, unstaking transfers assets from the silo to the receiver. Hence unstaking does not trigger hooks relating to blacklisting.

Impact

Blacklisted users can still unstake from the vault. If a user starts cooldown to redeem assets then becomes blacklisted and fully restricted, they can still unstake assets. This goes against the intended behavior of the protocol, as fully restricted users should not be able to unstake.

Recommendations

Add a check in `unstake` to ensure the caller `msg.sender` is not fully restricted. Also, `redistributeLockedAmount` needs to be reworked to have functionality to transfer the locked assets from the silo.

Remediation

This issue has been acknowledged by AxisFDN, and a fix was implemented in commit [a1165763](#) ↗.

3.5. Fully restricted addresses can still deposit

Target	StakedAxisUSD		
Category	Coding Mistakes	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

In StakedAxisUSD, addresses can be soft restricted or full restricted through `addToBlacklist`, which adds `SOFT_RESTRICTED_STAKER_ROLE` or `FULL_RESTRICTED_STAKER_ROLE`. Soft-restricted addresses cannot stake, and full-restricted addresses should not be able to transfer, stake, or unstake.

```

/// @notice The role which prevents an address to stake
bytes32 private constant SOFT_RESTRICTED_STAKER_ROLE
    = keccak256("SOFT_RESTRICTED_STAKER_ROLE");
/// @notice The role which prevents an address to transfer, stake, or unstake.
    The owner of the contract can redirect address staking balance if an
    address is in full restricting mode.
bytes32 private constant FULL_RESTRICTED_STAKER_ROLE
    = keccak256("FULL_RESTRICTED_STAKER_ROLE");

```

However, full-restricted addresses can still stake, since the `_deposit` function only checks for `SOFT_RESTRICTED_STAKER_ROLE`.

```

/**
 * @dev Deposit/mint common workflow.
 * @param caller sender of assets
 * @param receiver where to send shares
 * @param assets assets to deposit
 * @param shares shares to mint
 */
function _deposit(address caller, address receiver, uint256 assets,
    uint256 shares)
    internal
    override
    nonReentrant
    notZero(assets)
    notZero(shares)
{
    if (hasRole(SOFT_RESTRICTED_STAKER_ROLE, caller) ||

```

```
hasRole(SOFT_RESTRICTED_STAKER_ROLE, receiver)) {  
    revert OperationNotAllowed();  
}  
super._deposit(caller, receiver, assets, shares);  
_checkMinShares();  
}
```

Impact

Fully restricted addresses can still deposit assets, as long as the receiver is not also blacklisted. This goes against the intended behavior.

Recommendations

Add a check in `_deposit` to ensure the caller does not have the `FULL_RESTRICTED_STAKER_ROLE` role.

Remediation

This issue has been acknowledged by AxisFDN, and a fix was implemented in commit [b5193776](#).

3.6. Per-block mint/redeem caps enable same-block denial of service via executor ordering

Target	AxisUSDMinting		
Category	Business Logic	Severity	Medium
Likelihood	Low	Impact	Medium

Description

AxisUSDMinting enforces per-block mint and redeem limits via `belowMaxMintPerBlock` / `belowMaxRedeemPerBlock` modifiers. Only addresses with `MINTER_ROLE` / `REDEEMER_ROLE` can call `mint` / `redeem`, and they execute off-chain signed user orders on chain.

A well-funded user can repeatedly submit mint and then redeem off-chain orders sized near `maxMintPerBlock` / `maxRedeemPerBlock`. If the `MINTER_ROLE` / `REDEEMER_ROLE` executors include these orders early each block, they consume the entire per-block limit. Later, same-block orders will revert due to the per-block limit, denying other users that block's service.

This grief is safe for the user, as funds are not at risk while deposited or redeemed through normal flows, but the user needs to pay gas while submitting the off-chain orders.

```
modifier belowMaxMintPerBlock(uint256 mintAmount) {
    if (mintedPerBlock[block.number] + mintAmount > maxMintPerBlock) {
        revert MaxMintPerBlockExceeded();
    }
    _;
}
```

```
function mint(Order calldata order, Route calldata route,
    Signature calldata signature)
    ...
    belowMaxMintPerBlock(order.xyusd_amount)
{
    ...
    mintedPerBlock[block.number] += order.xyusd_amount;
    ...
}
```

Impact

A user can temporarily cause a denial of service for `mint` and `redeem` by spamming off-chain orders that executors run first each block, exhausting the per-block limits. This can persist for a limited time (bounded by inclusion costs and executor policy) and may be especially harmful during time-sensitive protocol events.

Recommendations

By design, the current guard enforces this behavior. Either

- formally accept and document the constraint (including operational/user impacts), or
- remove the guard to avoid the denial of service.

Please note that off-chain address filtering is insufficient as a single actor can sybil across multiple addresses to bypass address-based filters and continue the denial of service.

Remediation

AxisFDN has acknowledged this issue and provided the following statement:

This DoS vector is acknowledged and acceptable. Protocol-controlled `MINTER_ROLE/REDEEMER_ROLE` addresses provide operational flexibility to mitigate through off-chain order prioritization and monitoring policies.

3.7. Lack of comprehensive tests

Target	N/A		
Category	Code Maturity	Severity	Medium
Likelihood	N/A	Impact	Medium

Description

The contracts have unit tests for most functions as well as a few fuzzing tests. However, not all code branches are covered by tests, as detailed in the threat model (section [5.1](#)). We recommend a more comprehensive testing suite, as Findings [3.3.1](#), [3.4.1](#), and [3.5.1](#) could have been prevented with further testing.

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. 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.

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 your 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 seems contradictory, given the time investment to create and maintain tests. To expand upon that, 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 something 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 indicator that the existing functionality most likely was not broken by your change to the code.

Impact

Insufficient tests led to many bugs and findings, which could have been prevented had there been sufficient testing. Not all code branches or scenarios were covered, increasing the risk of bugs in these regions.

Recommendations

Improve the current testing suite by ensuring tests cover 100% of code branches. Fuzzing tests should also be written especially for the ERC-4626 vault, to ensure its behaviour functions as expected. This includes cases such as direct transfers into the vault, and depositing and withdrawing various amounts should be covered.

Remediation

AxisFDN has acknowledged this issue and provided the following statement:

We will be adding more tests for better coverage for point 3.7 over time.

3.8. No method to withdraw Ether in AxisUSDMinting

Target	AxisUSDMinting		
Category	Business Logic	Severity	Medium
Likelihood	Low	Impact	Low

Description

AxisUSDMinting has a method to receive Ether but does not use Ether in any calls and does not have a method to withdraw the Ether.

```
/**
 * @notice Fallback function to receive ether
 */
receive() external payable {
    emit Received(msg.sender, msg.value);
}
```

Impact

This results in received Ether being locked permanently. The contract is also not upgradable, so there is no method to retrieve the transferred Ether.

Recommendations

Add a method for the admin to be able to withdraw Ether.

Remediation

This issue has been acknowledged by AxisFDN, and a fix was implemented in commit [0d2368b8](#).

3.9. Missing check to prevent silo from being blacklisted

Target	StakedAxisUSD, StakedAxisUSDV2		
Category	Business Logic	Severity	Medium
Likelihood	Low	Impact	Low

Description

In StakedAxisUSDV2, when cooldown is enabled and during withdrawal, assets are temporarily held at silo and users are able to unstake after the cooldown period has elapsed. However, StakedAxisUSD implements a blacklist with two restriction modes, soft restriction and full restriction. If the silo was fully restricted, this would prevent users from withdrawing and unstaking assets.

Impact

If the silo becomes blacklisted, users will not be able to cool down assets and shares or unstake from StakedAxisUSDV2.

Recommendations

Add checks to ensure the silo never becomes blacklisted.

Remediation

This issue has been acknowledged by AxisFDN, and a fix was implemented in commit [e7b0d60c](#).

3.10. Admin could get blacklisted through transferAdmin or grantRole

Target	AxisUSDMinting		
Category	Business Logic	Severity	Low
Likelihood	Low	Impact	Low

Description

There is the invariant that the admin of AxisUSDMinting should never be blacklisted, as they may need to perform tasks that involve receiving assets, which may get restricted through blacklisting.

However, when the admin is transferred to another address through transferAdmin, there is no check. Hence, the new admin could already be blacklisted and then get transferred the admin. Additionally, the admin can call grantRole and give themselves the blacklisted roles.

Impact

If the admin becomes blacklisted, functions such as redistributeLockedAmount may fail if the admin is the recipient. However, this is minimal impact as the admin can easily remove their blacklisted role.

Recommendations

Add a check in transferAdmin to ensure the new admin is not blacklisted, and add checks in grantRole to ensure that if the admin is the target, then roles being granted should not be the blacklisted roles.

Remediation

This issue has been acknowledged by AxisFDN, and a fix was implemented in commit [50fd6fcf](#).

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. Front-running concerns for StakedAxisUSDV2

StakedAxisUSDV2 inherits from StakedAxisUSD, which uses a blacklist for addresses with two restriction modes, soft and full restriction. Fully restricted users essentially cannot perform any actions relating to the staking vault, including transfers, staking, and unstaking. However, there is always the possibility that a user with shares who is about to become fully restricted will observe the transaction to fully restrict them in the mempool and front-run the transaction to transfer assets to another address, preventing their funds from being locked. This is by inherent design and is not easily preventable but should be kept in mind as a possibility when blacklisting addresses.

4.2. Custody wallet configuration and redemption liquidity

When users redeem Axis USD for a whitelisted collateral token in AxisUSDMinting, the contract pays the beneficiary from collateral it holds. By contrast, during minting the user's collateral is routed to custody wallets configured off-chain. If those custody wallet allowlists include addresses other than the minting contract itself, collateral may be sent elsewhere, leaving the contract underfunded at redeem time and unable to pay beneficiaries in full.

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.

5.1. Module: AxisUSDMinting.sol

Function: `confirmDelegatedSigner(address _delegatedBy)`

This function allows the delegated address to confirm delegation.

Inputs

- `_delegatedBy`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** This address must have requested to set the caller as the delegated signer.
 - **Impact:** The address that delegated the signing to the caller.

Branches and code coverage

Intended branches

- Check the caller was delegated by the specified `_delegatedBy` address.
 - ☐ Test coverage
- Set the caller as the delegated signer of the `_delegatedBy` address. Delegation status should be set to accepted.
 - ☒ Test coverage

Negative behavior

- Revert if the caller was not delegated by the specified `_delegatedBy` address.
 - ☐ Negative test

Function: `mint(Order order, Route route, Signature signature)`

This function mints Axis USD given whitelisted collateral. It transfers the collateral to custody wallets and Axis USD to the beneficiary. Collateral should be worth the same as the Axis USD being minted. Collateral pricing is calculated off chain.

Inputs

- order
 - **Control:** Whitelisted users have control over the fields `order_type`, `expiry`, `collateral_asset`, and `collateral_amount`.
 - **Constraints:** The order must be signed by a benefactor (whitelisted user). The `expiry` is constrained as quotes from RFQ off chain are valid for only 15 minutes. Also, `collateral_asset` must be whitelisted.
 - **Impact:** Which collateral to use, how much collateral to transfer, and the time frame the order is valid for.
- route
 - **Control:** No control for the user — only controlled by the caller with `MINTER_ROLE`.
 - **Constraints:** Route total ratio must sum to `ROUTE_REQUIRED_RATIO`.
 - **Impact:** The addresses the collateral is transferred to, with ratios determining how much collateral to allocate each address.
- signature
 - **Control:** No control for the user — only controlled by the caller with `MINTER_ROLE`.
 - **Constraints:** N/A.
 - **Impact:** The order signature used to verify the order was signed by the benefactor.

Branches and code coverage

Intended branches

- Check that the function caller has role `MINTER_ROLE`.
 - ☒ Test coverage
- Check that the order mint amount will not cause `mintedPerBlock` to exceed `maxMintPerBlock`.
 - ☒ Test coverage
- Check that the order type is mint.
 - ☐ Test coverage
- Verify the order is valid.
 - ☒ Test coverage
- Verify the route is valid.
 - ☒ Test coverage
- Verify the nonce has not already been used.

- ☒ Test coverage
 - Verify that collateral is in the whitelist.
- ☒ Test coverage
 - Transfer collateral from the benefactor to the given route addresses.
- ☒ Test coverage
 - Mint Axis USD to the beneficiary.
- ☒ Test coverage

Negative behavior

- Revert if the caller does not have role MINTER_ROLE.
- ☒ Negative test
 - Revert if the order mint amount will cause mintedPerBlock to exceed maxMintPerBlock.
- ☒ Negative test
 - Revert if the order type is not mint.
- ☒ Negative test
 - Revert if the order has expired.
- ☒ Negative test
 - Revert if the order does not have a correct signature.
- ☒ Negative test
 - Revert if the route is not valid.
- ☒ Negative test
 - Revert if the nonce has already been used.
- ☒ Negative test
 - Revert if the collateral asset is not whitelisted.
- ☒ Negative test

Function: `redeem(Order order, Signature signature)`

Given an order and a signature signed by either the benefactor or a delegated signer, this function burns tokens and sends the corresponding amount of specified collateral tokens to the beneficiary.

Inputs

- order

- **Control:** Whitelisted users have control over the fields `order_type`, `expiry`, `collateral_asset`, and `collateral_amount`.
- **Constraints:** The order must be signed by a benefactor (whitelisted user). The expiry is constrained as quotes from RFQ off chain are valid for only 15 minutes. Also, `collateral_asset` must be whitelisted.
- **Impact:** Which collateral to use, how much collateral to transfer, and the time frame the order is valid for.
- signature
 - **Control:** Provided by the benefactor.
 - **Constraints:** Signature should be signed by the benefactor or an address delegated to by the benefactor.
 - **Impact:** The order signature used to verify the order was signed by the benefactor.

Branches and code coverage

Intended branches

- Check that the caller has role `REDEEMER_ROLE`.
 - ☒ Test coverage
- Check that the redemption will not cause the `redeemedPerBlock` limit to be exceeded.
 - ☒ Test coverage
- Check that the order type is a redeem order.
 - ☒ Test coverage
- Verify the order is valid.
 - ☒ Test coverage
- Verify the nonce has not already been used.
 - ☒ Test coverage
- Increment `redeemedPerBlock` by the amount of Axis USD being redeemed.
 - ☐ Test coverage
- Burn the order amount of Axis USD from the benefactor.
 - ☒ Test coverage
- Verify that collateral is in the whitelist.
 - ☐ Test coverage
- Transfer the order's amount of collateral to the beneficiary.
 - ☒ Test coverage

Negative behavior

- Revert if the caller does not have role REDEEMER_ROLE.
 - ☒ Negative test
- Revert if the redemption will cause the redeemedPerBlock limit to be exceeded.
 - ☒ Negative test
- Revert if the order is not a redeem order.
 - ☒ Negative test
- Revert if the order is not valid.
 - ☒ Negative test
- Revert if the nonce has already been used.
 - ☒ Negative test
- Revert if the benefactor does not have enough Axis USD to burn.
 - ☐ Negative test
- Revert if the collateral is not in the whitelist.
 - ☐ Negative test

Function: `removeDelegatedSigner(address _removedSigner)`

This function removes a delegated signer for the caller.

Inputs

- `_removedSigner`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** The address to undelegate.

Branches and code coverage

Intended branches

- Undelegate the specified `_removedSigner` address by setting its delegation status with the caller to rejected.
 - ☒ Test coverage

Negative behavior

- N/A.

Function: `setDelegatedSigner(address _delegateTo)`

This function delegates a signer for mint/redeem order signing. It sets the delegated address to pending status for the caller.

Inputs

- `_delegateTo`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** The signer to delegate signing to for the caller.

Branches and code coverage

Intended branches

- Set the delegated address to pending status for the caller.

☒ Test coverage

Negative behavior

- N/A.

Function: `transferToCustody(address wallet, address asset, uint256 amount)`

This function allows a caller with `COLLATERAL_MANAGER_ROLE` to transfer any ERC-20 asset to whitelisted custody wallets.

Inputs

- `wallet`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Cannot be the zero address. Must be one of the whitelisted custody wallets.
 - **Impact:** The custody wallet address to transfer assets to.
- `asset`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** ERC-20 assets to transfer.
- `amount`

- **Control:** Fully controlled by the caller.
- **Constraints:** N/A.
- **Impact:** Amount of ERC-20 tokens to transfer.

Branches and code coverage

Intended branches

- Check the caller has the role COLLATERAL_MANAGER_ROLE.
☒ Test coverage
- Check the custody wallet to transfer to is not the zero address and is whitelisted.
☐ Test coverage
- Transfer amount of ERC-20 asset tokens to wallet.
☒ Test coverage

Negative behavior

- Revert if caller does not have role COLLATERAL_MANAGER_ROLE.
☒ Negative test
- Revert if specified custody wallet is the zero address or is not whitelisted.
☐ Negative test

Function: `verifyNonce(address sender, uint256 nonce)`

This function checks that a nonce is unused for a sender using a bitmap scheme.

Inputs

- sender
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Must correspond to the account whose nonce is being consumed/checked.
 - **Impact:** Selects the bitmap row (`_orderBitmaps[sender]`).
- nonce
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Must be nonzero and must not already be marked used. Slot is `uint64(nonce) >> 8`; bit is `1 << uint8(nonce)`.
 - **Impact:** Determines which slot and bit are checked within the sender's bitmap.

Branches and code coverage

Intended branches

- Nonce is unused and nonzero.

☒ Test coverage

Negative behavior

- `nonce == 0`.

☐ Negative test

- Nonce is already used (`bitmap & bit != 0`).

☐ Negative test

Function: `verifyOrder(Order order, Signature signature)`

This function verifies an EIP-712 signed order. It computes the typed-data hash, recovers the signer, enforces the signer is the benefactor or an accepted delegate, performs basic field checks, and returns the order hash.

Inputs

- `order`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** `order.beneficiary != address(0)`, `order.collateral_amount > 0`, `order.xyusd_amount > 0`, and `block.timestamp <= order.expiry`. Other fields (e.g., `nonce`, `collateral_asset`) are not validated here and are checked downstream.
 - **Impact:** Defines the data hashed and authorizing context.
- `signature`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Must be a valid ECDSA signature over the EIP-712 hash of `order` for either `order.benefactor` or an ACCEPTED delegated signer.
 - **Impact:** Proves authorization — invalid or expired signatures cause reverts.

Branches and code coverage

Intended branches

- Order is signed by an accepted delegate or the benefactor.

☒ Test coverage

- Order is valid and nonexpired.

☒ Test coverage

Negative behavior

- Signer is neither benefactor nor an accepted delegate.

☐ Negative test

- `block.timestamp > order.expiry`.

☐ Negative test

- `order.beneficiary == address(0)`.

☐ Negative test

- `order.collateral_amount == 0` or `order.xyusd_amount == 0`.

☐ Negative test

Function call analysis

- `this.hashOrder(order) -> MessageHashUtils.toTypedDataHash(this.getDomainSeparator(), keccak256(this.encodeOrder(order)))`
 - **What is controllable?** order fields are caller-controlled — the domain separator depends on chainid and address(this).
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `ECDSA.recover(taker_order_hash, signature.signature_bytes)`
 - **What is controllable?** signature.signature_bytes and order (which defines the hash).
 - **If the return value is controllable, how is it used and how can it go wrong?** Returns recovered signer — mismatch fails authorization check.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `verifyRoute(Route route)`

This function validates a collateral distribution route.

Inputs

- route
 - **Control:** Fully controlled by the caller.
 - **Constraints:** `route.addresses.length == route.ratios.length` and `> 0`; every `route.addresses[i] != address(0)` and is in `_custodianAddresses`; every `route.ratios[i] > 0`; and `sum(route.ratios) == ROUTE_REQUIRED_RATIO`.
 - **Impact:** Determines if a route may be used by upstream flows (e.g., mint).

Branches and code coverage

Intended branches

- Valid route (all checks pass).

☒ Test coverage

Negative behavior

- `addresses.length != ratios.length`.

☐ Negative test

- Contain noncustodian or `address(0)`.

☐ Negative test

- Contain zero ratio.

☐ Negative test

- Ratios sum `!= ROUTE_REQUIRED_RATIO`.

☐ Negative test

5.2. Module: StakedAxisUSDV2.sol

Function: `cooldownAssets(uint256 assets)`

This function cools down the assets after the cooldown has finished.

Inputs

- assets
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Amount of assets that have cooled down.

Branches and code coverage

Intended branches

- Check if the cooldown duration is on.
 - ☒ Test coverage
- Check if the assets are greater than the max withdrawal.
 - ☒ Test coverage
- Set the cooldown end and underlying amount.
 - ☒ Test coverage
- Call `_withdraw`.
 - ☒ Test coverage

Negative behavior

- Revert transaction if the cooldown duration is off.
 - ☒ Negative test
- Revert transaction if the assets are greater than the max withdrawal.
 - ☒ Negative test

Function: `cooldownShares(uint256 shares)`

This function cools down the shares after the cooldown has finished.

Inputs

- `shares`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Amount of shares cooled down.

Branches and code coverage

Intended branches

- Check if the cooldown duration is on.
 - ☒ Test coverage
- Check if the shares are greater than the max redeem.
 - ☒ Test coverage

- Set the cooldown end and underlying amount.

☒ Test coverage

- Call `_withdraw`.

☒ Test coverage

Negative behavior

- Revert transaction if the cooldown duration is off.

☒ Negative test

- Revert transaction if the shares are greater than the max redeem.

☒ Negative test

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

This function calls the ERC-4626 `redeem` function with a check for cooldown duration.

Inputs

- `shares`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Amount of shares redeemed.
- `receiver`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Address of the receiver.
- `_owner`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Address of the owner.

Branches and code coverage

Intended branches

- Check if the cooldown duration is off.

☒ Test coverage

- Call `super.redeem`.

☒ Test coverage

Negative behavior

- Revert transaction if the cooldown duration is on.

☒ Negative test

Function: `unstake(address receiver)`

This function unstakes the staked amount after the cooldown has finished.

Inputs

- `receiver`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Address of the receiver.

Branches and code coverage

Intended branches

- Reset the cooldown and call `silo.withdraw` if the cooldown is finished.

☒ Test coverage

Negative behavior

- Revert transaction if the cooldown is not finished.

☒ Negative test

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

This function calls the ERC-4626 withdraw function with a check for cooldown duration.

Inputs

- `assets`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Amount of assets withdrawn.

- `receiver`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Address of the receiver.
- `_owner`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Address of the owner.

Branches and code coverage

Intended branches

- Check if the cooldown duration is off.
 - ☒ Test coverage
- Call `super.withdraw`.
 - ☒ Test coverage

Negative behavior

- Revert transaction if the cooldown duration is on.
 - ☒ Negative test

5.3. Module: StakedAxisUSD.sol

Function: `transferInRewards(uint256 amount)`

This function is used to transfer rewards from the rewarder into this contract.

Inputs

- `amount`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Amount of rewards transferred into this contract.

Branches and code coverage

Intended branches

- Check if the caller is the rewarder.
 - ☒ Test coverage
- Update vesting amount.
 - ☒ Test coverage
- Transfer assets from the rewarder to this contract.
 - ☒ Test coverage

Negative behavior

- Revert transaction if caller is not the rewarder.
 - ☒ Negative test
- Revert transaction if unvested amount is greater than zero.
 - ☒ Negative test

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

This function overrides the ERC-4626 `_deposit` function to check if the caller or receiver has been soft restricted.

Inputs

- `caller`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Must not be soft restricted.
 - **Impact:** Address of the caller.
- `receiver`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Must not be soft restricted.
 - **Impact:** Address of the receiver.
- `assets`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Amount of assets deposited.
- `shares`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.

- **Impact:** Amount of shares minted.

Branches and code coverage

Intended branches

- Check if the caller or receiver is soft restricted.
☒ Test coverage
- Call `super._deposit`.
☒ Test coverage
- Check if the minimum shares is violated.
☒ Test coverage

Negative behavior

- Revert transaction if the caller or receiver is soft restricted.
☒ Negative test
- Revert transaction if the minimum shares is violated.
☒ Negative test

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

This function overrides the ERC-20 `_update` function to check if the `from` or `to` address has been fully restricted.

Inputs

- `from`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Must not be fully restricted.
 - **Impact:** Address of the `from` address.
- `to`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Must not be fully restricted.
 - **Impact:** Address of the `to` address.
- `value`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.

- **Impact:** Amount of tokens transferred.

Branches and code coverage

Intended branches

- Check if the `from` or `to` address is fully restricted.
☒ Test coverage
- Call `super._update`.
☒ Test coverage

Negative behavior

- Revert transaction if the `from` or `to` address is fully restricted.
☒ Negative test

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

This function overrides the ERC-4626 `_withdraw` function to check if the caller, receiver, or owner has been fully restricted.

Inputs

- `caller`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Must not be fully restricted.
 - **Impact:** Address of the caller.
- `receiver`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Must not be fully restricted.
 - **Impact:** Address of the receiver.
- `_owner`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** Must not be fully restricted.
 - **Impact:** Address of the owner.
- `assets`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.

- **Impact:** Amount of assets transferred out.
- shares
 - **Control:** Fully controlled by the caller.
 - **Constraints:** N/A.
 - **Impact:** Amount of shares burned.

Branches and code coverage

Intended branches

- Check if the caller, receiver, or owner is fully restricted.
 - ☒ Test coverage
- Call `super._withdraw`.
 - ☒ Test coverage
- Check if the minimum shares is violated.
 - ☒ Test coverage

Negative behavior

- Revert transaction if the caller, receiver, or owner is fully restricted.
 - ☒ Negative test
- Revert transaction if the minimum shares is violated.
 - ☒ Negative test

Function: `transferInRewards(uint256 _rewardsAmount)`

This function transfers XYUSD rewards from this distributor contract into the staking vault. Only the operator may call it.

Inputs

- `_rewardsAmount`
 - **Control:** Fully controlled by the caller (operator).
 - **Constraints:** Must be $\leq$ `XYUSD_TOKEN.balanceOf(address(this))`.
 - **Impact:** Amount of XYUSD attempted to be transferred into the staking vault and accounted for as rewards.

Branches and code coverage

Intended branches

- Called by the operator with sufficient balance.
 - ☐ Test coverage
- Called by a non-operator.
 - ☐ Test coverage
- Called by the operator with insufficient balance.
 - ☐ Test coverage

Negative behavior

- The caller is not operator.
 - ☐ Negative test

Function call analysis

- `this.XYUSD_TOKEN.balanceOf(address(this))`
 - **What is controllable?** The external token balance could be increased by direct transfer, but this will not harm the protocol or users.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.STAKING_VAULT.transferInRewards(_rewardsAmount)`
 - **What is controllable?** `_rewardsAmount` is provided by the operator.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters or does other unusual control flow?** Sub function reverts, no state is mutated, and no reentrancy is possible.

6. Assessment Results

During our assessment on the scoped Axis contracts, we discovered 10 findings. Two critical issues were found. One was of high impact, four were of medium impact, and three were of low impact.

The next steps we recommend for improving the codebase are to strengthen test coverage, with a focus on adding multi-step tests that validate sequential operations. While the overall code complexity is moderate, the current tests are too limited and fail to cover all branches, which led to several bugs being uncovered (see Finding [3.7. 7](#)). Expanding the suite with broader coverage will greatly improve security, provide stronger assurance of correctness, and increase confidence in the reliability of the protocol.

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.