Aztec Coinissuer

Security Review

Solo review by:
R0bert, Lead Security Researcher

November 12, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A security review is a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While the review endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that a security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Aztec Labs was founded in 2017, and has a team of +50 leading zero-knowledge cryptographers, engineers, and business experts. Aztec Labs is developing its namesake products: AZTEC (a privacy-first L2 on Ethereum) and NOIR (the universal ZK language).

From Nov 2nd to Nov 3rd the security researchers conducted a review of aztec-packages on commit hash bd31f776. The review focused on the following scope:

- l1-contracts/src/governance/CoinIssuer.sol

A total of **8** issues were identified:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	0	0	0
Gas Optimizations	1	1	0
Informational	7	5	2
Total	**8**	**6**	**2**

3 Findings

3.1 Gas Optimization

3.1.1 Cache budget to cut duplicate `SLOAD`

Severity: Gas Optimization

Context: CoinIssuer.sol#L75-L77

Description: In `mint` the contract reads `cachedBudget` twice: once for the allowance check and again for the subtraction. That costs two SLOADs per mint even though the value is unchanged between those operations:

```
function mint(address _to, uint256 _amount) external onlyOwner {
  _updateBudgetIfNeeded();

  require(_amount <= cachedBudget, Errors.CoinIssuer__InsufficientMintAvailable(cachedBudget, _amount));
  cachedBudget -= _amount;

  ASSET.mint(_to, _amount);
}
```

Caching the value in a local variable and reusing it for both the require payload and the subtraction collapses the work into a single load.

Recommendation: Store the allowance in a stack variable before the require, use it in the revert payload, and update the storage slot from that cached value (optionally inside an `unchecked` block):

```
  function mint(address _to, uint256 _amount) external onlyOwner {
    _updateBudgetIfNeeded();

-   require(_amount <= cachedBudget, Errors.CoinIssuer__InsufficientMintAvailable(cachedBudget, _amount));
-   cachedBudget -= _amount;
+   uint256 budget = cachedBudget;
+   require(_amount <= budget, Errors.CoinIssuer__InsufficientMintAvailable(budget, _amount));
+   unchecked {
+     cachedBudget = budget - _amount;
+   }

    ASSET.mint(_to, _amount);
  }
```

Aztec Labs: Fixed in PR 18188.

R0bert: Fix verified.

3.2 Informational

3.2.1 `CoinIssuer` cap exceeds 100 percent

Severity: Informational

Context: CoinIssuer.sol#L50

Description: The constructor stores the provided annual rate without checking its bounds, so deployments can set an allowance higher than the entire supply. Because `_getNewBudget` multiplies `totalSupply()` by this unchecked percentage, any cap above `1e18` (100%) yields an annual budget greater than the whole token supply. In that scenario the owner can legally mint more than the existing supply in a single year.

```
constructor(IMintableERC20 _asset, uint256 _annualPercentage, address _owner) Ownable(_owner) {
  ASSET = _asset;
  NOMINAL_ANNUAL_PERCENTAGE_CAP = _annualPercentage;
  DEPLOYMENT_TIME = block.timestamp;
  cachedBudgetYear = 0;
  cachedBudget = _getNewBudget();
}

function _getNewBudget() private view returns (uint256) {
  return ASSET.totalSupply() * NOMINAL_ANNUAL_PERCENTAGE_CAP / 1e18;
}
```

Recommendation: Consider if this behaviour is wanted and if not, validate the annual cap when it is provided (and in any future updates) so that values greater than `1e18` are rejected, ensuring the computed budget never exceeds the full supply.

Aztec Labs: Added a comment in PR 18188 that >100% is accepted.

R0bert: Verified.

3.2.2 External minters bypass annual cap

Severity: Informational

Context: CoinIssuer.sol#L78

Description: The yearly allowance is fixed at the start of each 365 day window, so `CoinIssuer` assumes it is the lone source of inflation. If another authorized minter increases the total supply during the window, the cached budget stays unchanged and the additional tokens sit outside the tracked allowance. When `_getNewBudget` runs at the year boundary it recalculates the next allowance from the now larger supply, but the prior year has already exceeded the intended inflation limit relative to the supply snapshot used. This means any concurrent minter can silently blow past the yearly cap, undermining the contract's primary monetary control for that year.

```
function mint(address _to, uint256 _amount) external onlyOwner {
  _updateBudgetIfNeeded();
  require(_amount <= cachedBudget, Errors.CoinIssuer__InsufficientMintAvailable(cachedBudget, _amount));
  cachedBudget -= _amount;
  ASSET.mint(_to, _amount);
}

function _updateBudgetIfNeeded() private {
  uint256 currentYear = _yearSinceGenesis();
  if (cachedBudgetYear < currentYear) {
    cachedBudgetYear = currentYear;
    cachedBudget = _getNewBudget();
    emit BudgetReset(currentYear, cachedBudget);
  }
}
```

The impact is that supply minted outside `CoinIssuer` can push real inflation well above the targeted ceiling for an entire year before any recalibration occurs.

Recommendation: Ensure `CoinIssuer` is the only address with minting privileges on the asset, or modify the asset so all minting must flow through this contract.

Aztec Labs: Added a comment in PR 18188 that we alternative ways to mint can bypass budget.

R0bert: Verified.

3.2.3 Owner lacks asset minter role risk

Severity: Informational

Context: CoinIssuer.sol#L78

Description: `CoinIssuer` forwards mint requests directly to the underlying token and assumes ownership implies minting privilege. Many ERC20s, including the repository's own `TestERC20`, gate minting behind an explicit minter role so the call to `ASSET.mint` reverts unless that role has been granted. Deployers who only transfer token ownership to `CoinIssuer` without also adding it as an authorized minter end up with a contract that fails every mint attempt, silently breaking issuance.

```
function mint(address _to, uint256 _amount) external onlyOwner {
  _updateBudgetIfNeeded();
  require(_amount <= cachedBudget, Errors.CoinIssuer__InsufficientMintAvailable(cachedBudget, _amount));
  cachedBudget -= _amount;
  ASSET.mint(_to, _amount);
}
```

`IMintableERC20` implementers typically require `addMinter(address(this))` or similar before minting succeeds, so relying on `acceptTokenOwnership` alone does not guarantee liveness.

Recommendation: Merely informative. Ensure the `CoinIssuer` contract has the required minting permissions over the `ASSET` token.

Aztec Labs: Added a comment in PR 18188 that the coinissuer must be a minter, either with the role or using the ownership if owner can mint.

R0bert: Verified.

3.2.4 Zero initial supply bricks minting

Severity: Informational

Context: CoinIssuer.sol#L54

Description: Deployment caches the first-year allowance by calling `_getNewBudget`, so a token with zero supply at that moment leaves `cachedBudget` stuck at zero. Every `mint` call checks `_amount <= cachedBudget`, causing the contract to revert on any nonzero mint and permanently locking the issuer until some other minter increases the supply. This is only hinted at in the comments, meaning a missed seeding step bricks the contract's liveness.

```
constructor(IMintableERC20 _asset, uint256 _annualPercentage, address _owner) Ownable(_owner) {
  ASSET = _asset;
  NOMINAL_ANNUAL_PERCENTAGE_CAP = _annualPercentage;
  DEPLOYMENT_TIME = block.timestamp;
  cachedBudgetYear = 0;
  cachedBudget = _getNewBudget();
}

function _getNewBudget() private view returns (uint256) {
  return ASSET.totalSupply() * NOMINAL_ANNUAL_PERCENTAGE_CAP / 1e18;
}
```

The impact is total loss of minting functionality when deployers forget to seed the token supply before constructing `CoinIssuer`.

Recommendation: Fail fast if the supply is zero (and in any future initializer) by checking `totalSupply() > 0` during deployment, or enforce an operational checklist/test that seeds the token supply before deploying CoinIssuer so the first-year budget is nonzero.

Aztec Labs: Fixed in PR 18188. We did slightly different here and had a check instead on the budget in the constructor. Namely as that would catch both the totalSupply being 0 and the case where the rate is 0. We think that would better cover it.

R0bert: Fix verified.

3.2.5 `CoinIssuer` contract is limited to a single asset

Severity: Informational

Context: CoinIssuer.sol#L49

Description: `CoinIssuer` hardcodes one ERC20 by storing `ASSET` as an immutable constructor parameter. All minting, budgeting, and events operate on that single token, so deployments need one `CoinIssuer` per asset. In ecosystems where governance manages multiple tokens, this increases operational overhead and duplicates code that could share rate logic across assets.

```
IMintableERC20 public immutable ASSET;

constructor(IMintableERC20 _asset, uint256 _annualPercentage, address _owner) Ownable(_owner) {
  ASSET = _asset;
  NOMINAL_ANNUAL_PERCENTAGE_CAP = _annualPercentage;
  // ...
}
```

Supporting multiple assets from a single contract (for example by mapping asset addresses to budgets and rates) would make the issuer more flexible for treasury operations that span several tokens.

Recommendation: If multi-asset support is desired, redesign the storage layout around mappings keyed by asset address (tracking supply snapshots, caps, and budgets per asset) and expose mint functions that

accept the target token; otherwise document that a separate `CoinIssuer` instance must be deployed for each ERC20.

Aztec Labs: Acknowledged.

R0bert: Acknowledged.

3.2.6 CoinIssuer does not use a double transfer ownership pattern

Severity: Informational

Context: CoinIssuer.sol#L48

Description: `CoinIssuer` inherits `Ownable`, so transferring control relies on a single `transferOwnership` call. The file already imports `Ownable2Step` to interact with assets that use that pattern, yet the issuer itself does not benefit from the pending-owner handshake. In high-governance environments a one-call transfer makes it easier to accidentally hand control to the wrong address or lose custody if a transaction is misconfigured, even though the contract is designed for deliberate, high-stakes minting authority.

```
contract CoinIssuer is ICoinIssuer, Ownable {
  IMintableERC20 public immutable ASSET;
  // ...
}
```

Using two-step ownership would align the issuer's control surface with the safer pattern already adopted by the underlying asset and reduce operational errors during ownership rotations.

Recommendation: Switch the inheritance to `Ownable2Step` (or add equivalent pending-owner flow) so ownership transfers require both `transferOwnership` and `acceptOwnership`, preventing misfires and mirroring the expected operational safeguards for mint authority.

Aztec Labs: Acknowledged. We are using the `Ownable` instead of `Ownable2Step` since we want the owner to be one contract during setup, where it can properly accept token etc, and then hand it over to the governance. Where we cannot make the governance just accept early as it need to go through a full proposal.

R0bert: Acknowledged.

3.2.7 `CoinIssuer`/`ERC20` edge cases

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `CoinIssuer` assumes the underlying token behaves like a plain OZ ERC20, yet many real deployments use variants that invalidate those assumptions. The helper below hard-casts the token to `Ownable2Step`, the budget math trusts raw `totalSupply()`, and minting proceeds without verifying the token's semantics:

```
function acceptTokenOwnership() external onlyOwner {
  Ownable2Step(address(ASSET)).acceptOwnership();
}

function mint(address _to, uint256 _amount) external onlyOwner {
  _updateBudgetIfNeeded();
  require(_amount <= cachedBudget, Errors.CoinIssuer__InsufficientMintAvailable(cachedBudget, _amount));
  cachedBudget -= _amount;
  ASSET.mint(_to, _amount);
}

function _getNewBudget() private view returns (uint256) {
  return ASSET.totalSupply() * NOMINAL_ANNUAL_PERCENTAGE_CAP / 1e18;
}
```

When the `ERC20` deviates from that model, multiple ”weird” edge cases appear:

1. Ownership mismatches: tokens using single-step Ownable, AccessControl, or no ownership at all make `acceptTokenOwnership()` revert.

2. External minters, rebasing, or other dynamic `totalSupply()`: the yearly budget freezes at the year boundary, so minting elsewhere (or rebases and burns) means the real inflation can far exceed the nominal cap for that year.
3. Fee-on-transfer, burn-on-transfer, or blacklists: the budget tracks `totalSupply()` but circulating supply may drop due to fees/burns and transfers to blacklisted/zero addresses can revert because `CoinIssuer` does not guard `_to`.
4. Exotic mint semantics: some tokens mint less than the requested `_amount`, meaning the cap no longer corresponds to the intended percentage.
5. Non-18-decimal UX: the rate uses a 1e18 fixed-point percentage, so operators can easily misconfigure caps when the asset has unusual decimals.
6. Pausable or upgradeable tokens: pausability can silently brick `CoinIssuer` even with remaining budget and upgrades might change mint or supply math without notice.
7. Extreme supplies: `totalSupply() * NOMINAL_ANNUAL_PERCENTAGE_CAP` can overflow before division, bricking year resets.

Recommendation: Merely informative. Consider documenting these risks.

Aztec Labs: Fixed in PR 18188.

R0bert: Fix verified.