Zellic

Prepared for
Christopher Goes
Anoma Foundation

Prepared by
Kritsada Dechawattana
Seunghyeon Kim
Zellic

July 17, 2025

Anoma Token

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 Anoma Token 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 10

2.5. Project Timeline 11

3. Detailed Findings **11**

3.1. Overlapping execution window between `claim` and `burnUnclaimedTokens` functions 12

3.2. Ambiguous operator precedence in bitwise comparison 14

4. Threat Model **15**

4.1. Module: ForeignReserveV1.sol 16

4.2. Module: TokenDistributor.sol 17

4.3. Module: XanV1.sol 23

5. Assessment Results 30

5.1. Disclaimer 31

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 Anoma Foundation from July 10th to July 14th, 2025. During this engagement, Zellic reviewed Anoma Token'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 potential bugs in the voting system that could affect governance or protocol behavior?
- Are there any risks or known issues related to the usage of an upgradable proxy contract in this context?
- Does the current locking mechanism introduce any unintended side effects, such as preventing expected transfers or interactions with external applications relying on standard ERC-20 behavior?
- Could external applications be affected if they assume that all tokens returned by ERC-20 `balanceOf` are fully transferable, when in fact, some balances may be locked or restricted?

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

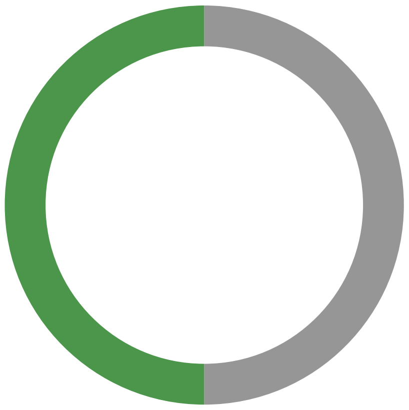
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Anoma Token contracts, we discovered two findings. No critical issues were found. One finding was of low impact and the other finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	1
⬜ Informational	1



2. Introduction

2.1. About Anoma Token

Anoma Foundation contributed the following description of Anoma Token:

> The Anoma token is a standard ERC20 token contract with a fixed distribution which can be upgraded (to arbitrary new logic) with a metagovernance mechanism based on quorum approval voting and a fast-track council.

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.

2.3. Scope

The engagement involved a review of the following targets:

Anoma Token Contracts

Type	Solidity
Platform	EVM-compatible

Target	token-distributor
Repository	https://github.com/anoma/token-distributor ↗
Version	`1a8ed882db93a9263097d06c2d392987606a01d0`

Programs	`TokenDistributor.sol`
	`interfaces/*`
	`libs/*`

Target	token
Repository	https://github.com/anoma/token ↗
Version	`856c38dd77d777783c4b0f7010419ef1b99a0daa`

Programs	`XanV1.sol`
	`ForeignReserveV1.sol`
	`interfaces/*`
	`libs/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 0.6 person-weeks. The assessment was conducted by two consultants over the course of three calendar days.

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 ↗

The following consultants were engaged to conduct the assessment:

Kritsada Dechawattana
Engineer
kritsada@zellic.io ↗

Seunghyeon Kim
Engineer
seunghyeon@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 10, 2025	Kick-off call
July 10, 2025	Start of primary review period
July 14, 2025	End of primary review period

3. Detailed Findings

3.1. Overlapping execution window between `claim` and `burnUnclaimedTokens` functions

Target	TokenDistributor		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

There is a condition with overlapping time logic that enables both `claim` and `burnUnclaimedTokens` to be callable at the exact end time. This can result in a race condition where one function being called may break or revert the other.

```
function claim(Allocation.Data calldata allocation, bytes32[] calldata proof)
    external payable override {

    [...]

@>  if (_END_TIME < currentTime) {
        revert EndTimeInThePast();
    }
    [...]

}

function burnUnclaimedTokens() external override {
@>  if (Time.timestamp() < _END_TIME) {
        revert EndTimeInTheFuture();
    }

    _XAN.burn(_XAN.balanceOf(address(this)));
}
```

Impact

If a user attempts to claim tokens exactly at the end time and the `burnUnclaimedTokens` function is called first (within the same block), their claim will fail due to the token balance being burned. This can lead to several issues: User Claims May Fail Unfairly: Users who submit valid claims at the final moment may be front-run or invalidated by the burn transaction, even though they meet the time condition. Potential for Griefing or Exploitation: Attackers may deliberately race transactions at the end time to prevent other users from claiming tokens, especially in high-value or airdrop scenarios. Automation Risks: If automated bots or scripts are used to execute burns or claims, they may unintentionally interfere with one another, causing failed executions or unnecessary gas usage.

Recommendations

To prevent this race condition and ensure predictable, fair contract behavior, the `claim` and `burnUnclaimedTokens` functions should not be allowed to overlap at the same timestamp. The execution windows must be mutually exclusive to avoid unexpected reverts. Consider allowing only one function to be called at the exact end time.

Remediation

This issue has been acknowledged by Anoma Foundation, and a fix was implemented in commit `db6b5555` ↗.

3.2. Ambiguous operator precedence in bitwise comparison

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

Description

In the TokenDistributor contract, the following line appears in the `_isClaimed` function:

```
claimed = claimedWord & mask == mask;
```

While this line is syntactically and semantically correct in Solidity — due to to Solidity's operator precedence rules — it may lead to misinterpretation by developers who are more familiar with other languages such as C or JavaScript. In those languages, the equality operator (`==`) has higher precedence than the bitwise `AND` operator (`&`), which could cause confusion during code reviews or future maintenance.

This style relies on implicit knowledge of Solidity's operator precedence, which may reduce code readability and clarity.

Impact

There is no functional or security impact from the current implementation, as the expression behaves as intended in Solidity. However, the lack of explicit parentheses could increase the cognitive load for readers and introduce potential misunderstanding during refactoring or auditing.

Additionally, there are behavioral changes regarding the evaluation order of expressions when compiling the contract through an intermediate representation ("IR"). You can learn more about these changes in **Solidity IR-based Codegen Changes ↗**.

Recommendations

Update the line to make the precedence explicit, improving clarity:

```
claimed = (claimedWord & mask) == mask;
```

This change is stylistic but improves code readability and reduces the chance of misinterpretation.

Remediation

This issue has been acknowledged by Anoma Foundation, and a fix was implemented in commit `ea65d763` ↗.

4. 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.

4.1. Module: ForeignReserveV1.sol

Function: `execute(address target, uint256 value, bytes calldata data)`

This function enables the owner (the XanV1 contract) to execute arbitrary calls after an upgrade for a seamless upgrade process with reentrant protection.

The client added negative tests in the commit `88a21af5d4111662e515ef411b81ddd42ca1f98d` ↗

Inputs

- `target`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The target contract to call.
- `value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This contract must have sufficient Ether for attachment.
  - **Impact**: The Ether attached to this call.
- `data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The calldata to be sent for this call.

Branches and code coverage

Intended branches

- The function accepts execution with Ether.
  - ☑ Test coverage
- The function accepts execution without Ether.

- ☑ Test coverage

Negative behavior

- The function reverts if not called by the owner.
    - ☑ Test coverage
- The function reverts if the target contract reverts.
    - ☑ Test coverage
- The function reverts when reentrant.
    - ☑ Test coverage

4.2. Module: TokenDistributor.sol

Function: `allocationTreeRoot()`

This function returns the root of the Merkle tree used for verifying allocations.

Function: `burnUnclaimedTokens()`

This function is designed to burn all unclaimed tokens from the distributor contract.

Branches and code coverage

Intended branches

- The current time is after the end of the distribution period.
    - ☑ Test coverage
- The function burns all unclaimed tokens.
    - ☑ Test coverage

Negative behavior

- The function reverts if the current time is before the end of the distribution period.
    - ☑ Negative test

Function: `claim(Allocation.Data allocation, byte[32][] proof)`

This function allows users to claim tokens by providing a valid allocation and its corresponding Merkle proof. It first verifies the authenticity of the allocation using the proof against the Merkle root. If the allocation is valid, the specified tokens — both locked and unlocked — are distributed to the recipient defined in the allocation data. If the verification fails or the allocation is otherwise invalid, the function reverts.

Inputs

- `allocation`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The allocation must be valid and exist in the allocation tree.
  - **Impact**: The allocation to be claimed.
- `proof`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The proof must be valid and correspond to the allocation.
  - **Impact**: The proof to verify the allocation's validity.

Branches and code coverage

Intended branches

- The function checks whether the current timestamp is earlier than the claim period's `startTime`. If so, the function reverts, as claims are not allowed before the distribution begins.
  - ☑ Test coverage
- The function ensures the current time is not later than the `endTime`. If the claim period has already ended, the function reverts to prevent late claims.
  - ☑ Test coverage
- The function checks whether the given `allocation.index` has already been claimed. If it has, the function reverts to enforce single-use allocations.
  - ☑ Test coverage
- The function verifies the validity of the allocation using the provided Merkle proof. If the proof does not match the allocation or the root, the function reverts to prevent unauthorized claims.
  - ☑ Test coverage
- The function checks that the fee sent with the transaction (`msg.value`) exactly matches the `allocation.fee` field. If there is a mismatch, the function reverts to ensure correct fee handling.
  - ☑ Test coverage
- After marking the allocation as claimed and emitting an event, the function attempts to forward the fee (if any) to the _FOREIGN_RESERVE contract. If this transfer fails, the function reverts to ensure no loss of funds.
  - ☑ Test coverage

- If the allocation includes a locked token amount, the function calls `_XAN.transferAndLock()` to lock the specified amount of tokens for the recipient.
  - ☑ Test coverage
- If the allocation includes an unlocked token amount, the function transfers the specified tokens directly to the recipient using `_XAN.safeTransfer()`.
  - ☑ Test coverage

Negative behavior

- The function must prevent any claim attempt before the distribution period has started. Claims made before `_START_TIME` are rejected to enforce the schedule.
  - ☑ Negative test
- The function must reject claims made after the distribution period has ended (`_END_TIME`), ensuring that late or expired claims are invalid.
  - ☑ Negative test
- The function must prevent double-claiming by checking if a given allocation has already been claimed. If the allocation was previously processed, the function reverts.
  - ☑ Negative test
- The function must reject claims with invalid Merkle proofs, ensuring that only users with a valid and verifiable allocation can claim tokens.
  - ☑ Negative test
- The function must reject transactions where the attached fee does not match the expected `allocation.fee`, preventing overpayment or underpayment.
  - ☑ Negative test
- The function must ensure that fee forwarding to the _FOREIGN_RESERVE contract succeeds. If the transfer fails, the function reverts to protect the integrity of the process.
  - ☑ Negative test
- The function must not attempt to transfer or lock tokens if the allocation specifies zero values for both locked and unlocked, but this is safely handled by the conditionals.
  - ☑ Negative test

Function: `constructor(byte[32] allocationRoot, uint48 startTime, uint48 endTime, address governanceCouncil)`

This function is the constructor for the TokenDistributor contract. It initializes the contract with the provided parameters.

Inputs

- `allocationRoot`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The Merkle root of the allocation tree.
- `startTime`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be before the `END_TIME`.
    - **Impact**: The start time of the claim period.
- `endTime`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be after the `START_TIME`.
    - **Impact**: The end time of the claim period.
- `governanceCouncil`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be a nonzero address.
    - **Impact**: The address of the governance council.

Branches and code coverage

Intended branches

- The function verifies that the `startTime` is strictly less than the `endTime`. If the `startTime` is equal to or greater than the `endTime`, the transaction is reverted to prevent an invalid claim period range.
    - ☑ Test coverage
- The function checks that the `startTime` is not set in the past relative to the current block timestamp. If the `startTime` is earlier than the current time, the function reverts to avoid enabling retroactive claims.
    - ☑ Test coverage
- The function ensures that the `endTime` is set in the future. If the `endTime` is equal to or earlier than the current timestamp, the constructor reverts to prevent initialization of an already expired claim period.
    - ☑ Test coverage
- The function confirms that the `governanceCouncil` address is not the zero address. If a zero address is provided, the function reverts to avoid assigning a critical role to an invalid address.
    - ☑ Test coverage

Negative behavior

- The function must prevent initialization when the `startTime` is equal to or greater than the `endTime`, as this would define an invalid or negative duration for the claim window.
  - ☑ Negative test
- The function must prevent deployment if the `startTime` is already in the past. This ensures that the claim window begins strictly in the future and that no claims are possible before contract deployment.
  - ☑ Negative test
- The function must reject any attempt to set the `endTime` in the past or at the current time, which would make the claim period instantly expire or be unusable.
  - ☑ Negative test
- The function must reject a zero address as the `governanceCouncil` to ensure the role is assigned to a valid, functioning account that can fulfill governance responsibilities.
  - ☑ Negative test

Function: `foreignReserve()`

This function returns the address of the foreign reserve contract.

Function: `isClaimed(uint256 index)`

This function checks if an allocation at a specific index has been claimed.

Inputs

- `index`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index to check for claiming status.

Function: `token()`

This function returns the address of the token being distributed.

Function: `_isClaimed(uint256 index)`

This function checks whether an index is claimed or not.

Inputs

- `index`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The index to check.

Branches and code coverage

Intended branches

- The function checks if the index is claimed.
    - ☑ Test coverage

Function: `_setClaimed(uint256 index)`

This function sets an index in the Merkle tree as claimed. It is intended to be called internally by the claim function within the contract.

Inputs

- `index`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The index to mark as claimed.

Branches and code coverage

Intended branches

- The function sets the claimed bit for the given index.
    - ☑ Test coverage

Function: `_verifyProof(Allocation.Data allocation, byte[32][] proof)`

This function verifies the inclusion proof of an allocation in the allocation Merkle tree.

Inputs

- `allocation`

  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The allocation data to be verified.
- `proof`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The proof must be a valid Merkle proof.
  - **Impact**: The Merkle proof to verify the allocation.

Branches and code coverage

Intended branches

- The function returns true when the proof is valid.
  - ☑ Test coverage

4.3. Module: XanV1.sol

Function: `cancelCouncilUpgrade()`

This function allows the council to cancel their currently scheduled upgrade.

Branches and code coverage

Intended branches

- The function resets the scheduled upgrade by updating `councilData.scheduledImpl = address(0)` and `councilData.scheduledEndTime = 0`.
  - ☑ Test coverage
- The function emits the `CouncilUpgradeCancelled` event.
  - ☑ Test coverage

Negative behavior

- The function reverts if no council upgrade is scheduled.
  - ☑ Test coverage

Function: `cancelVoterBodyUpgrade()`

This function allows users to cancel a scheduled upgrade that did not receive the most votes and has not reached the required quorum.

Branches and code coverage

Intended branches

- The function checks that the delay period is over before canceling.
  - ☑ Test coverage
- The function emits the `VoterBodyUpgradeCancelled` event.
  - ☑ Test coverage
- The function resets the scheduled upgrade by updating the `votingData.scheduledImpl = address(0)` and `votingData.scheduledEndTime = 0`.
  - ☑ Test coverage

Negative behavior

- The function reverts if no voter-body upgrade is scheduled.
  - ☑ Test coverage
- The function reverts if the delay period has not ended.
  - ☑ Test coverage
- The function reverts if the upgrade is most voted and has reached the required quorum.
  - ☑ Test coverage

Function: `castVote(address proposedImpl)`

This function enables users to vote the proposed upgrade address with their locked tokens.

Inputs

- `proposedImpl`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: New votes must be higher than the previous votes.
  - **Impact**: The proposed upgrade address to vote for.

Branches and code coverage

Intended branches

- The function increases the `ballot.votes[voter]` for updating the amount of votes of the caller.
  - ☑ Test coverage
- The function increases the `ballot.totalVotes` for updating the total vote of this

`proposedImpl`.

  - ☑ Test coverage

- The function emits the `VoteCast` event.
  - ☑ Test coverage
- The function updates the most votes to this `proposedImpl` if the total vote is higher than the previous amount of most votes.
  - ☑ Test coverage

Negative behavior

- The function reverts if the new vote is lower than the previous vote plus one.
  - ☑ Test coverage

Function: `scheduleCouncilUpgrade(address impl)`

This function enables the council to schedule an upgrade if there is no voter-body upgrade that received the most votes.

Inputs

- `impl`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The proposed upgrade address that needs to be scheduled.

Branches and code coverage

Intended branches

- The function updates the `councilData.scheduledImpl` to promote this `impl` to be scheduled.
  - ☑ Test coverage
- The function updates the `councilData.scheduledEndTime` associated with the delay duration.
  - ☑ Test coverage
- The function emits the `CouncilUpgradeScheduled` event.
  - ☑ Test coverage

Negative behavior

- The function reverts if a voter-body upgrade could be scheduled.
  - ☑ Test coverage
- The function reverts if a council upgrade is already scheduled.
  - ☑ Test coverage

Function: `scheduleVoterBodyUpgrade()`

This function allows users to schedule the upgrade that received the most votes and has reached the required quorum. It always cancels a council upgrade if the council has proposed an upgrade.

Branches and code coverage

Intended branches

- The function updates the `votingData.scheduledImpl` to promote the most voted implementation to be scheduled.
  - ☑ Test coverage
- The function updates the `votingData.scheduledEndTime` associated with the delay duration.
  - ☑ Test coverage
- The function emits the `VoterBodyUpgradeScheduled` event.
  - ☑ Test coverage
- The function cancels the council upgrade if the council has proposed an upgrade.
  - ☑ Test coverage

Negative behavior

- The function reverts if another upgrade is scheduled by the voter body.
  - ☑ Test coverage
- The function reverts if the most voted implementation has not reached quorum.
  - ☑ Test coverage

Function: `transferAndLock(address to, uint256 value)`

This function enables users to transfer unlocked tokens to another user and then lock tokens immediately for future use as voting power on a proposed implementation.

Inputs

- `to`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Value needed to lock for voting power.
- `value`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Users must have sufficient unlocked tokens to lock.
    - **Impact**: Value needed to transfer and lock for voting power.

Branches and code coverage

Intended branches

- The function checks the caller has sufficient unlocked tokens to transfer.
    - ☑ Test coverage
- The function transfers tokens to the specific address before locking the `to` account tokens.
    - ☑ Test coverage
- The function increases the `data.lockedSupply` to track the locked balances associated with the current implementation.
    - ☑ Test coverage
- The function increases the `data.lockedBalances[account]` to track the locked balances associated with the account.
    - ☑ Test coverage
- The function emits the lock event after proceeding with the lock.
    - ☑ Test coverage

Negative behavior

- The function reverts if the call has insufficient unlocked tokens.
    - ☑ Test coverage

Function: `vetoCouncilUpgrade()`

This function allows users to veto the council-scheduled upgrade if there is any upgrade that received the most votes and reached the required quorum.

Branches and code coverage

Intended branches

- The function resets the scheduled upgrade by updating the `councilData.scheduledImpl = address(0)` and `councilData.scheduledEndTime = 0`
    - ☑ Test coverage
- The function emits the `CouncilUpgradeVetoed` event.
    - ☑ Test coverage

Negative behavior

- The function reverts if no council upgrade is scheduled.
    - ☑ Test coverage
- The function reverts if the most voted upgrade has not reached quorum or the minimum locked supply.
    - ☑ Test coverage

Function: `_authorizeUpgrade(address newImpl)`

This function is triggered to authorize the required conditions and update the state before performing the upgrade.

Inputs

- `newImpl`
    - **Control**: Full control.
    - **Constraints**: The `newImpl` address must be scheduled before.
    - **Impact**: The new implementation of this contract.

Branches and code coverage

Intended branches

- The function resets the scheduled upgrade by updating `votingData.scheduledImpl = address(0)` and `votingData.scheduledEndTime = 0` if it is a voter-body upgrade.
    - ☑ Test coverage
- The function resets the scheduled upgrade by updating `councilData.scheduledImpl = address(0)` and `councilData.scheduledEndTime = 0` if it is a council upgrade.
    - ☑ Test coverage

Negative behavior

- The function reverts if the upgrade is scheduled by both entities.
  - ☑ Test coverage
- The function reverts if the upgrade is not the most voted for a voter-body upgrade.
  - ☑ Test coverage
- The function reverts if the upgrade has not reached quorum or the minimum locked supply for a voter-body upgrade.
  - ☑ Test coverage
- The function reverts if the upgrade is not scheduled.
  - ☑ Test coverage
- The function reverts if a voter-body upgrade could be scheduled.
  - ☑ Test coverage
- The function reverts if the delay period has not ended.
  - ☑ Test coverage

Function: `_lock(address account, uint256 value)`

This function locks the unlocked tokens from an account for future use as voting power on proposing an upgrade.

Inputs

- `account`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This account must have sufficient unlocked tokens to lock.
  - **Impact**: The account address needed to lock the tokens.
- `value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Account must have sufficient unlocked tokens to lock.
  - **Impact**: The value needed to lock for voting power.

Branches and code coverage

Intended branches

- The function increases the `data.lockedSupply` for tracking the locked balances associated with the current upgrade.

- ☑ Test coverage
- The function increases the `data.lockedBalances[account]` for tracking the locked balances associated with the account.
  - ☑ Test coverage
- The function emits the `Locked` event after proceeding with the lock.
  - ☑ Test coverage

Negative behavior

- The function reverts if the account has insufficient unlocked tokens.
  - ☑ Test coverage

5. Assessment Results

During our assessment on the scoped Anoma Token contracts, we discovered two findings. No critical issues were found. One finding was of low impact and the other finding was informational in nature.

5.1. 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.