Zellic

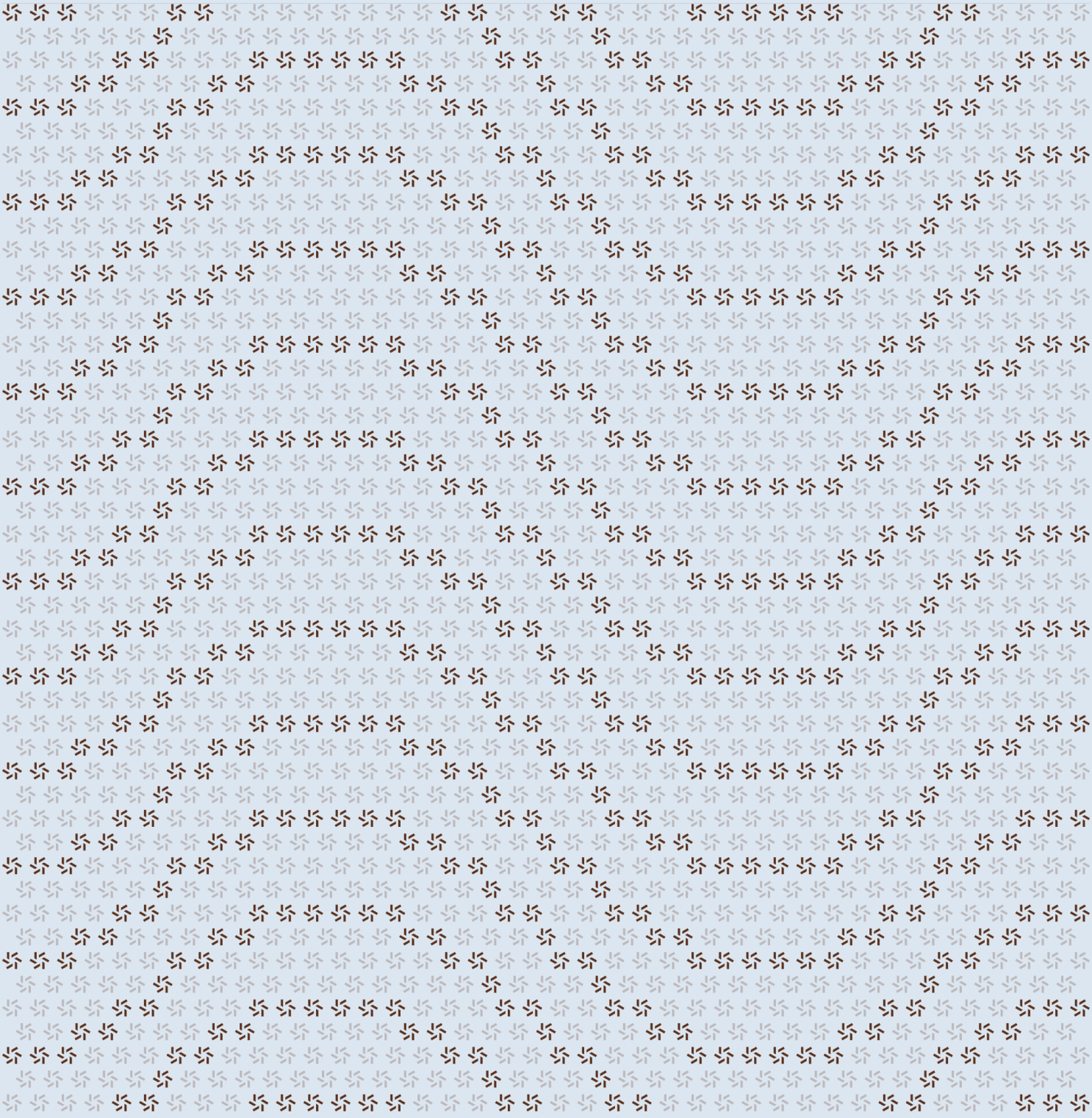
Prepared for
mprwa-dev
MatrixDock

Prepared by
Sunwoo Hwang
Varun Verma
Zellic

July 28, 2025

XAUm

Sui Move Application Security Assessment



Contents

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 MatrixDock from July 16th to July 21st, 2025. During this engagement, Zellic reviewed XAUm'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:

- Is there proper access control for minting and burning XAUm tokens?
- Is there proper access control for managing modules?
- Are there any security issues in mint requests?

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 XAUm modules, we discovered four findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

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

2. Introduction

2.1. About XAUm

MatrixDock contributed the following description of XAUm:

> Matrixdock Gold (XAUm) is a standardized token deployed on multiple chains, with a 1:1 peg to 1 troy oz. fine weight of high grade LBMA gold. The total supply of XAUm will always be equal to the amount of underlying assets stored in highly secured and reputable vaults.

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 modules.

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 modules itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

XAUm Modules

Type	Move
Platform	Sui
Target	RWA-Contracts
Repository	https://github.com/Matrixdock-RWA/RWA-Contracts ↗
Version	`0f6de9a8b56c3b2cc5d97f5177907bdc73292405`
Programs	`xaum-sui/packages/*.move`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.1 person-weeks. The assessment was conducted by two consultants over the course of four 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

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Sunwoo Hwang
Engineer
sunwoo@zellic.io

Varun Verma
Engineer
varun@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 16, 2025	Kick-off call
July 16, 2025	Start of primary review period
July 21, 2025	End of primary review period

3. Detailed Findings

3.1. Lack of 2-step ownership transfer

Target	mtoken.move		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

In the current design of ownership transfer, the owner calls both `request_transfer_ownership` and `execute_transfer_ownership` to transfer ownership. The `UpgradeCap` is also transferred to the new owner, so if the new owner address is the wrong address, the ownership cannot be revoked. The receiver address should be a valid address that can receive the ownership.

Impact

If the new owner address is the wrong address, the ownership cannot be revoked.

Recommendations

We recommend changing the design to implement a proper 2-step process where the new owner calls `execute_transfer_ownership` to claim the ownership and `UpgradeCap`.

Remediation

This issue has been acknowledged by MatrixDock, and a fix was implemented in commit c4890239 ↗.

3.2. Incorrect role check

Target	mtoken.move		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In the `revoke_set_revoker` function, it checks if the sender is the `operator`. While all functions with the `revoke_` prefix are only callable by the `revoker`, when setting the revoker, it would be more appropriate to check if the sender is the `owner`. This is because the `operator` is only responsible for operations related to minting and burning coins.

```
entry fun revoke_set_revoker<T>(state: &State<T>, req: SetRevokerReq, ctx:
    &TxContext) {
    check_version(state);
    check_operator(state, ctx);
```

Impact

The operator can revoke operations that exceed its privileges.

Recommendations

We recommend to check if the sender is the `owner` instead of `operator`.

Remediation

This issue has been acknowledged by MatrixDock, and a fix was implemented in commit `fbab42d6` ↗.

3.3. Timestamp Validation Permissiveness

Target	minter.move		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Current timestamp validation at `minter.move` L186:

```
assert!(now <= timestamp + DELAY_MAX, ...);
```

This check ensures that the current time is not after `timestamp + DELAY_MAX`, rejecting stale requests. However, it allows any future timestamp, meaning users can submit requests with timestamps significantly ahead of the current block time.

Impact

Permitting forward-dated timestamps may lead to inconsistencies if off-chain systems (e.g., price feeds or processors) assume timestamps reflect actual request time. In some cases, this could enable manipulative behavior or cause confusion in time-sensitive processing.

Recommendations

Consider enforcing a bounded timestamp range on both sides:

```
assert!(now >= timestamp && now <= timestamp + DELAY_MAX, ...);
```

This ensures that requests are neither stale nor prematurely timestamped. If future timestamps are expected, document this behavior for downstream systems.

Remediation

This issue has been acknowledged by MatrixDock and they have provided the following response:

> The backend performs further validation on the timestamp and thus not necessary here.

3.4. Budget Check Timing in Mint Workflow

Target	mtoken.move		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the minting flow, `request_mint_to` queues a mint request without checking whether the requested amount fits within `state.mint_budget`. That validation only occurs later during `execute_mint_to`. As a result, oversized requests can be accepted and stored on-chain, only to fail upon execution.

Impact

While this does not cause incorrect behavior, it introduces a gap between request acceptance and request viability. Malformed or oversized mint requests remain on-chain until explicitly executed and then revert, potentially leading to user confusion or unnecessary storage bloat.

Recommendations

Consider performing a budget check at the request phase:

```
assert!(amount <= state.mint_budget, EMintBudgetNotEnough);
```

This would provide earlier feedback and prevent the creation of unexecutable requests.

Remediation

This issue has been acknowledged by MatrixDock and they have provided the following response:

> This logic is consistent with the EVM version of the code and the result is expected.

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. Off-chain validation

Users can mint XAUm by calling the `request_to_mint` function. Several parameters require validation, including `transferred_token` and `for_token` matching, `preprice`, `slippage`, and `timestamp` values. However, the minter module does not perform these checks and only stores the transferred token and emits a `MintRequest` event.

```
public entry fun request_to_mint<T>(
    state: &State,
    transferred_token: &mut Coin<T>,
    for_token: address,
    amount: u64,
    preprice: u64,
    slippage: u64,
    timestamp: u64,
    clock: &Clock,
    ctx: &mut TxContext,
) {
```

This issue has been acknowledged by MatrixDock and they have provided the following response:

> Parameter validation is performed off-chain.
>
> The XAUm minting process follows these steps:
>
> 1. **User deposits USDC**: User calls `request_to_mint` function to deposit USDC
> 2. **Matrixdock server validation**:
>   - Validates that the from address's bound UID has passed KYC2
>   - Verifies USDC amount exceeds minimum threshold
>   - Compares user-provided pre-price with system timestamp price
> 3. **Validation outcome**:
>   - **Failure**: Matrixdock operator manually refunds USDC on-chain
>   - **Success**: Proceeds to next step
> 4. **Off-chain processing**: USDC converted to USD, physical gold purchased and vaulted

5. **Update mint budget**: Operator calls `change_mint_budget` to ensure minted XAUm doesn't exceed stocked gold
6. **Request mint**: Operator calls `request_mint_to`
7. **Execute mint**: After delay, operator calls `execute_mint_to`
8. **Mint completion**: XAUm minted to Matrixdock's address, then transferred to user

4.2. Component: Minter Module (`minter::minter`)

Description

Governs a mint and redeem gateway between stable-coin pool A and RWA pool B on Sui. It stores pool addresses, token whitelists and version data, enforces request timing, routes funds, and emits events for off-chain processors.

What it does

- **State management**: Keeps a single `State` object with owner, version, pool addresses, and two token-acceptance tables.
- **Admin control**: Owner-only functions to transfer ownership, migrate version, set pool addresses, and add or remove accepted tokens.
- **User flows**: `request_to_mint<T>` and `request_to_redeem<T>`.
    - Validate token against the relevant whitelist.
    - Enforce timestamp within delay and check coin balance.
    - Split the user coin and send the requested amount to the correct pool.
    - Emit `MintRequest` or `RedeemRequest` event.
- **View helpers**: constant-time getters for public state.

How it does it

Uses Move primitives such as `assert!`, `coin::split`, `transfer::public_transfer`, and `event::emit`. Generic type introspection converts the type parameter into an address for whitelist checks. Private helpers `check_version` and `check_owner` centralize validation logic.

Invariants

- `state.version` equals `VERSION` after every successful entry (except `migrate`, which sets it).
- Only `state.owner` may call admin entries; enforced by `check_owner`.

- `request_to_mint` accepts only tokens in `accepted_by_a`, `request_to_redeem` only tokens in `accepted_by_b`.
- `timestamp` in requests must be within fifty-nine seconds of `clock`.
- `amount` cannot exceed `transferred_token.value`.
- On success, the exact amount moves to the target pool and the caller's remaining balance is intact.

Test coverage

Cases covered:

- Initialization sets version to one, sets the owner, and sets empty whitelists.
- Owner updates pool addresses and whitelists; getters return expected values.
- Positive mint request with whitelisted USDT to pool A.
- Checks that balance is reduced, event fields are correct, and pool receives funds.
- Positive redeem request with whitelisted SUI to pool B (mirror of above).
- Mint with token not whitelisted triggers `EInvalidTokenForMint`.
- Mint with stale timestamp triggers `EInvalidTimestamp`.
- Mint after removing token from whitelist triggers `EInvalidTokenForMint`.

Cases not covered:

- Redeem with token not whitelisted (`EInvalidTokenForRedeem`).
- Insufficient coin balance (`EInsufficientBalance`).
- Owner-guard violations (`ENotOwner`).
- Upgrade path logic (`migrate`, `transfer_ownership` with `UpgradeCap`).

However, the client updated the test suite to support such cases in commit dd655f3 ↗.

Attack surface

- **User-supplied generic type `T`**: whitelist check aborts unaccepted tokens.
- **Timestamp spoofing**: allowed window limited to fifty-nine seconds.
- **Amount field**: `u64` checks and balance comparison block overflow.
- **UpgradeCap misuse**: wrong cap address aborts with `EUpgradeCapInvalid`.

External inputs are limited to coins, type, amount, and timestamp; each is checked before state mutation.

4.3. Component: MToken Module (`mtoken::mtoken`)

Description

Provides a fungible token on Sui with time-locked governance and fine-grained roles. It wraps `coin::create_regulated_currency_v2` and enforces owner, operator and revoker permissions with a configurable execution delay.

What it does

- **Token issuance and redemption** `request_mint_to` followed by `execute_mint_to` mints coins for a recipient and deducts the amount from a prefunded mint budget. `redeem` burns user coins and credits the budget.
- **Role management** For owner, operator and revoker through on-chain requests, execute and revoke cycles guarded by a global delay.
- **Parameter governance** For metadata, delay, role changes and ownership transfer, each is staged in a request object that becomes executable after the delay and then expires.
- **Compliance** Through a deny list that can block or unblock accounts and emits events for every action.
- **Safety rails** That enforce versioning, minimum delay, execution window validity and budget limits. All privileged calls check exact roles.

How it does it

Request objects carry an effective time set to the current time plus delay. Execution checks that the current time is within the allowed window. Helper functions check roles and budget arithmetic. All critical state lives in a single `State<T>` object keyed by the coin phantom type.

Invariants

- `state.version` always equals `VERSION`.
- `delay` is never below `MIN_DELAY`.
- Only the owner can create or update requests, only the operator can change budget, mint or redeem, and only the revoker can cancel requests.
- `mint_budget` cannot be negative and changes only by the exact minted or burned amount.
- A request executes only if the current time is at least the effective time and before it expires.
- Minting fails if it would exceed `mint_budget`.
- `TreasuryCap` and `DenyCap` never leave the `State` object except through controlled functions.

Test coverage

Cases covered:

- Deployment flow including coin creation and initial state verification.
- Version migration success and failure paths.
- Metadata updates with correct and incorrect roles.
- Ownership transfer request, execute and revoke including edge cases.
- Operator and revoker changes with full life-cycle tests.
- Delay changes respecting the minimum delay and timing rules.
- Mint-budget adjustments including overflow and underflow guards.
- Minting request, execute and revoke with budget and timing checks.
- Redemption by operator and rejection of non-operator attempts.
- Deny list block and unblock actions with role checks.
- Basic transfers of minted coins between users.

Cases not covered:

- Stress tests for rapid mint and burn sequences at budget limits.
- Deny list enforcement on transfers. However it is important to note that difficulty simulating epochs in SUI tests make such behavior harder to test for. `transfer_err_denied_src` attempts to test for this and is currently commented out.
- Extreme values for delay or amounts beyond the 64-bit range. In practice, this would be testing for runtime errors on overflow and underflow.

Attack surface

- User supplied addresses and amounts in mint, redeem and governance requests are checked for overflow and budget limits.
- Effective time manipulation is restricted to owner or operator actions and execution enforces the valid window.
- `UpgradeCap` misuse is prevented by verifying the package address before ownership transfer.
- Arithmetic errors are mitigated through safe arithmetic checks.
- Only the operator can modify the deny list and transfer rules.
- The generic type parameter is fixed at coin creation and cannot be changed later.

External actors can only request actions. Each privileged state change is delayed, role-gated, time-bounded and logged, limiting attack vectors to compromise of owner or operator credentials.

5. Assessment Results

During our assessment on the scoped XAUm modules, we discovered four findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining findings were 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.