Zellic

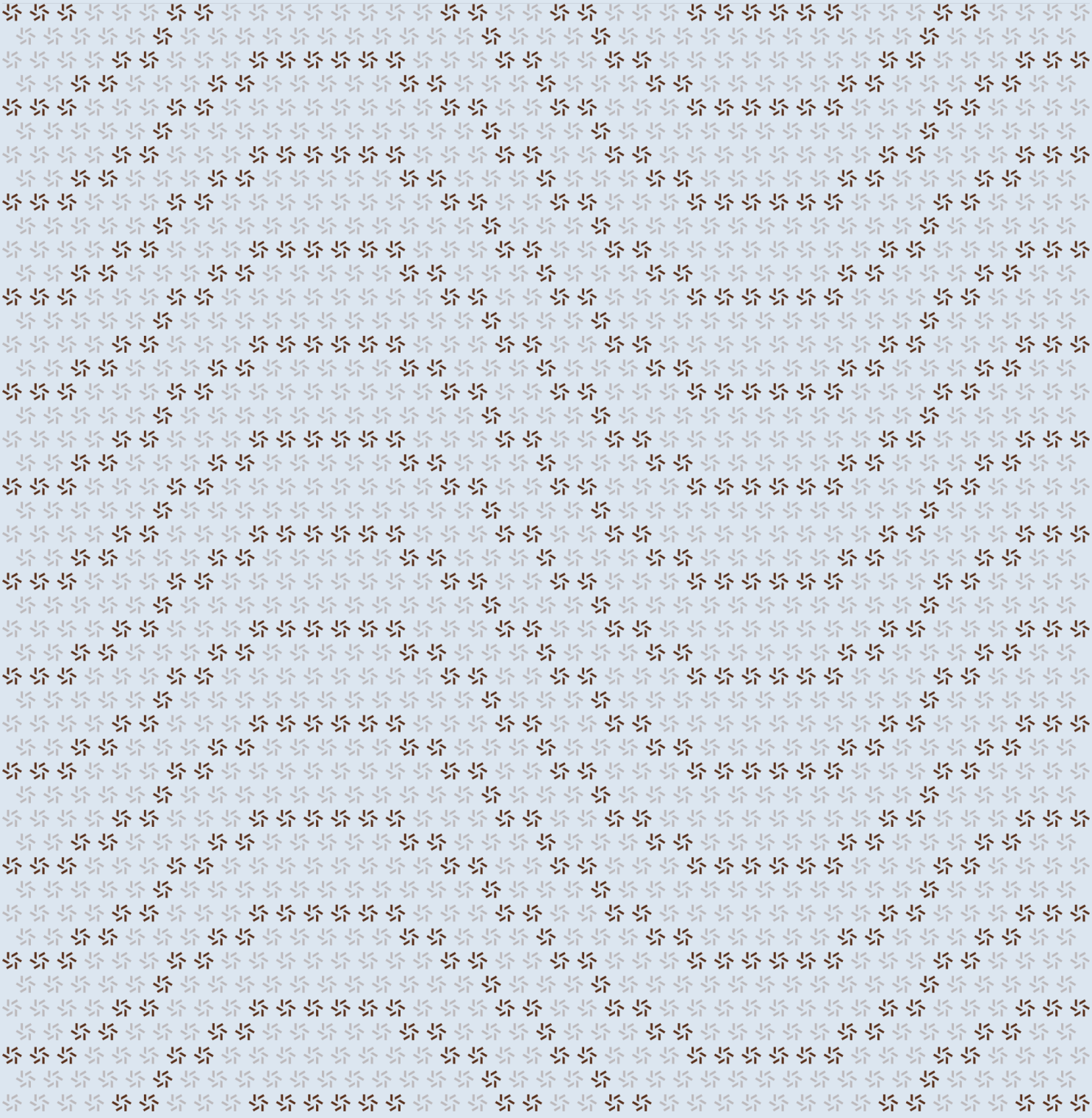
Prepared for
mprwa-dev
Matrixdock

Prepared by
Evan Hwang
Varun Verma
Zellic

March 21, 2026

XAGm

Sui Move Application 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 XAGm 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. The `UpgradeCap` can be stolen 11

3.2. Zero-amount mint/redeem request spam 13

4. Discussion **14**

4.1. New messenger cap can block OApp operations 15

4.2. Denylist handling in `cc_receive` 15

5. System Design **16**

5.1. Component: MToken 17

5.2. Component: Messenger OApp 18

5.3. Component: Minter 20

6. Assessment Results **21**

6.1. Disclaimer 22

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 March 17th to March 20th, 2026. During this engagement, Zellic reviewed XAGm'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:

- Does it follow the correct integration for LayerZero?
- Can an attacker bypass the blacklist mechanism?
- Can an attacker mint an arbitrary amount of tokens?
- Can an attacker steal ownership of the module?

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.

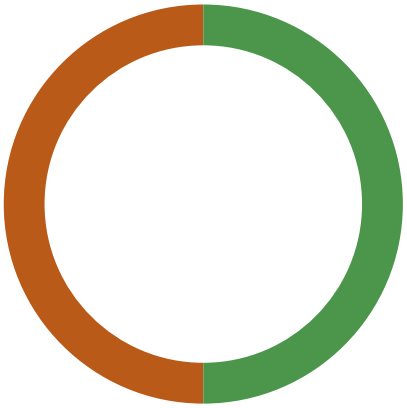
1.4. Results

During our assessment on the scoped XAGm modules, we discovered two findings. No critical issues were found. One finding was of high impact and one was of low impact.

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	1
Medium	0
Low	1
Informational	0



2. Introduction

2.1. About XAGm

Matrixdock contributed the following description of XAGm:

> Matrixdock Silver (XAGm) is a standardized token on blockchains, where 1 token initially represents one fine troy ounce of 99.9% purity LBMA-accredited Good Delivery silver. This representation will decrease slightly over time as custody fees accrue and is reflected in the ozPerToken daily value, which represents the amount of silver weight each token corresponds to.

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:

XAGm Modules

Type	move
Platform	Sui
Target	RWA-Contracts
Repository	https://github.com/Matrixdock-RWA/RWA-Contracts ↗
Version	`d30134ee11edbd2b56f330b02c3dfbf31368d66e`
Programs	`xagm-sui/**/sources/**/*.move`

2.4. Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Evan Hwang
Engineer
evan@zellic.io ↗

Varun Verma
Engineer
varun@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 17, 2026	Kick-off call
March 17, 2026	Start of primary review period
March 20, 2026	End of primary review period

3. Detailed Findings

3.1. The `UpgradeCap` can be stolen

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

Description

When transferring ownership, a `TransferOwnershipReq` is created along with the `UpgradeCap`. However, the `TransferOwnershipReq` is not bound to the type of state. As a result, anyone can call `revoke_transfer_ownership` with their own fake mToken, allowing them to steal the `UpgradeCap`.

```
entry fun revoke_transfer_ownership<T>(
    state: &State<T>,
    req: TransferOwnershipReq,
    ctx: &TxContext,
) {
    check_version(state);
    check_owner(state, ctx);
    let TransferOwnershipReq { id, upgrade_cap, .. } = req;
    transfer::public_transfer(upgrade_cap, state.owner); // @audit-issue: does
    not verify generic type T
    id.delete();
}
```

These type checks are missing in all revoke functions, so anyone can cancel the following operations:

- `SetOperatorReq`
- `SetRevokerReq`
- `SetDelayReq`
- `MintReq`

Impact

An attacker can steal the ownership of the module.

Recommendations

Add a generic type check in all revoke functions.

Remediation

This issue has been acknowledged by Matrixdock, and a fix was implemented in commit d9bc2f84 ↗.

3.2. Zero-amount mint/redeem request spam

Target	minter.move		
Category	Integration Risks	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The `request_to_mint` function never enforces `amount > 0`. With `amount == 0`, `balance >= amount` still holds and the flow can proceed, so an attacker pays only gas to emit arbitrary zero-value `MintRequest` / `RedeemRequest` events and overload indexers or off-chain settlement pipelines.

```
public fun request_to_mint<T, F>(
    state: &State,
    transferred_token: &mut Coin<T>,
    amount: u64,
    preprice: u64,
    slippage: u64,
    timestamp: u64,
    clock: &Clock,
    ctx: &mut TxContext,
) {
    check_version(state);
    let tn = type_name::with_defining_ids<T>();
    assert!(state.accepted_by_a.contains(tn), EInvalidTokenForMint);
    let now = clock.timestamp_ms() / 1000;
    assert!(now <= timestamp + DELAY_MAX, EInvalidTimestamp);
    let balance = transferred_token.value();
    assert!(balance >= amount, EInsufficientBalance);
    let out = coin::split<T>(transferred_token, amount, ctx);
```

Impact

While this is not a security issue, unbounded bogus events can degrade or flood off-chain consumers (settlement, monitoring, analytics) and waste operator time.

Recommendations

Add `assert!(amount > 0, ...)` at the start of each request entry point.

Remediation

This issue has been acknowledged by Matrixdock, and a fix was implemented in commit f8bd793f ↗.

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. New messenger cap can block OApp operations

In OApp, `MessengerCap` rotation is not supported safely. The mToken owner can mint a new cap and update the `MessengerCap` ID, which is used to verify the cross-chain call, while the OApp only installs a cap once and has no replacement path. After rotation, the OApp may still hold the old cap, so calls that pass it into mToken fail. That breaks both cross-chain sending and receiving in the code at the time of writing.

The Matrixdock team replied with the following:

> In the EVM version of MToken, we restrict cross-chain related methods so that only the Messenger contract can call them, by checking the messenger address. The MToken contract also stores a single messenger address, which cannot be arbitrarily changed. However, on Sui, we cannot check the `msg.sender` address. Therefore, we use the capability pattern to achieve a similar effect. To keep the behavior consistent with the EVM version, the Sui MToken contract allows only one valid `MessengerCap` object to exist at any given time. Issuing a new capability will invalidate the previous one.

4.2. Denylist handling in `cc_receive`

The function `cc_receive` only checks `coin::deny_list_v2_contains_current_epoch`. If the receiver lands on the denylist after the source burn but that check still passes, `public_transfer` will transfer the minted XAGm instead of storing it in the blocked tokens, so `blocked_tokens` stays empty and the claim path is never used.

The Matrixdock team replied with the following:

> This is the expected behavior and is by design.
>
> - If the recipient has been added to the deny list and the restriction is already in effect (as determined by `deny_list_v2_contains_current_epoch`), the tokens are held by the system. They can only be claimed after the address is removed from the deny list. This check exists so LayerZero cross-chain transactions can always succeed on the Sui side.
> - If the recipient has been added to the deny list but the restriction is not yet effec-

tive (as can be determined with `deny_list_v2_contains_next_epoch`), tokens are processed normally and transferred to the address. Because the address is already blacklisted, it can receive assets but is not allowed to send them.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: MToken

Description

What is the component responsible for? MToken is the core token-management module for XAGm on Sui. It owns the regulated token state, privileged roles, mint/redeem accounting, denylist integration, fee-rate–based OZ accounting, and the bridge-facing token / mint-budget primitives.

What does it do? It creates the regulated coin, stores the shared contract state, lets the operator request and execute minting, lets the operator redeem tokens, lets the operator manage the denylist, lets the owner manage roles and delay parameters, and exposes `cc_send_token`, `cc_send_mint_budget`, and `cc_receive` for the messenger integration.

How does it do it? The module stores `TreasuryCap` and `DenyCapV2` under the shared `State<T>` object via dynamic object fields. Sensitive admin actions use request objects plus an effective-time / TTL window. Cross-chain entry points require a `MessengerCap` whose object ID matches the ID currently stored in state. Mint and redeem flows also check an `expected_oz_per_token` value against the on-chain calculation derived from `annual_fee_rate`, `oz_per_token_base`, and elapsed full days.

Invariants

- The `mint_budget` should never become negative. Paths that spend it call `deduct_mint_budget`, which aborts if the remaining budget is too small.
- Delayed requests should only execute after their effective time and before request expiry.
- Owner-/operator-/revoker-gated functions should only be callable by the corresponding role.
- Cross-chain helpers should only accept the currently registered `MessengerCap`.
- The `annual_fee_rate` should never exceed `MAX_ANNUAL_FEE_RATE`, and a newly requested delay should stay within the configured minimum/maximum bounds.
- Mint and redeem should only proceed when the caller-supplied `expected_oz_per_token` matches the current on-chain value.

Test coverage

Cases covered

- Role checks for owner-, operator-, and revoker-gated flows
- Request/execute/revoke life cycles for ownership transfer, operator, revoker, delay, and mint
- Effective-time and expiry checks on delayed requests
- Mint/redeem behavior and mint-budget accounting
- Block/unblock behavior through the denylist
- Direct token transfer after minting
- Messenger-cap creation, replacement, and invalid-cap rejection
- The `cc_send_token`, `cc_send_mint_budget`, and `cc_receive` positive and negative paths
- Blocked-recipient handling in `cc_receive`
- Manual mint-budget send/receive helpers
- Fee-rate initialization and update, `oz_per_token`, `get_oz_amount`, and `expected_oz_per_token` mismatch checks
- Upgrade-cap initialization and migration checks
- Cross-state request-misuse proof of concepts using a different `State<T>` instance

Cases not covered

- Minting to a blocklisted recipient
- Long-horizon `oz_per_token` underflow behavior
- Interaction between pending requests and later delay changes
- Behavior around reissuing `MessengerCap` while older caps may still be held elsewhere

Attack surface

- The owner, operator, and revoker are all privileged trust boundaries. Compromise of any of those keys affects different parts of supply control and request cancellation.
- Also, `cc_receive` relies on the current messenger-cap holder and whatever authenticity checks the messenger/LayerZero layer applies before calling into MToken.
- Request objects are shared objects and are a sensitive area. The current test suite (at the time of writing) includes cross-state misuse proof of concepts, so request/state binding is an important audit surface.
- When a cross-chain receive targets a blocked address, the token is minted first and then returned as escrowed output instead of being transferred immediately.

5.2. Component: Messenger OApp

Description

What is the component responsible for? Messenger OApp is the LayerZero-facing bridge module. It owns the local OApp configuration, stores the MToken messenger capability, sends outbound bridge messages, receives inbound bridge messages, and holds blocked tokens that cannot be

transferred immediately.

What does it do? It lets the owner initialize and configure the OApp, set peers and enforced options, pause outbound sends, send token and mint-budget messages, confirm LayerZero sends, and process inbound LayerZero receives, and it lets users claim blocked tokens after they are no longer blocked.

How does it do it? The module stores LayerZero call/admin caps plus the optional `MessengerCap` inside a shared `State`. Outbound sends use the LayerZero two-step pattern and return a `SendContext` carrying the expected `call_id`. Inbound receives call into MToken's `cc_receive`, then either drop the result or accumulate blocked tokens in a per-receiver table.

Invariants

- Owner-only admin functions should remain owner-only.
- Outbound sends should fail while paused; inbound receives are intentionally not pause-gated.
- Receiver address length should match the configured length for a destination chain when such a length is configured.
- The `confirm_send` should only accept the `Call` paired with the matching `SendContext.call_id`.
- Blocked tokens should stay keyed by receiver and only be claimable through that receiver's transaction.

Test coverage

Cases covered

- Initial state checks
- Owner-only checks for messenger-cap initialization, OApp registration, OApp information updates, skip, peer configuration, enforced options, and pause
- Successful `send_token` and `send_mint_budget` flows
- Pause enforcement on outbound sends
- Receiver-length validation on outbound token sends
- Ownership transfer flow and upgrade/migration checks
- The `handle_cc_receive` blocked-token accumulation behavior
- Success and no-token failure paths of `claim_blocked_token`
- PTB builder / `lz_receive_info` generation

Cases not covered

- `confirm_send` with a mismatched `SendContext`
- Messenger-side behavior when no `mtoken_msg_cap` has been initialized
- Claim while the receiver is still blocked

- A full token-message receive flow through `lz_receive`
- Ordering/concurrency scenarios across multiple sends

Attack surface

- The owner can change peers, enforced options, pause state, and other OApp configuration. Misconfiguration or key compromise can redirect or halt bridge traffic.
- User-controlled send inputs include destination EID, receiver bytes, token/amount, and extra options. The local contract only enforces a configured receiver-length check; correctness beyond that depends on the wider bridge setup.
- Inbound message authenticity is outside this module's own logic and depends on LayerZero endpoint / peer validation before `lz_receive` is executed.
- Blocked tokens accumulate in local state and may remain there indefinitely if the receiver never becomes transferable again.
- Because receives are not pause-gated, a paused messenger still processes inbound messages.

5.3. Component: Minter

Description

What is the component responsible for? Minter is an intent-routing module for off-chain mint/redeem fulfillment. It accepts user deposits for buy or sell requests and emits structured events that an off-chain system can act on.

What does it do? It lets users call `request_to_mint` with an accepted payment token or `request_to_redeem` with an accepted RWA token, moves the requested amount to the configured pool account, and emits a `MintRequest` or `RedeemRequest` event containing the request parameters.

How does it do it? The owner configures pool addresses and token allowlists keyed by type name. Each request checks the relevant allowlist, rejects timestamps that are too old, verifies the caller has enough balance in the provided coin, splits out the requested amount, transfers that split coin to the configured pool account, and emits the event.

Invariants

- Only token types explicitly allowlisted for pool A should be usable for mint requests.
- Only token types explicitly allowlisted for pool B should be usable for redeem requests.
- The transferred amount should equal the requested amount because the code uses `coin::split`.
- Pool-account and allowlist configuration should remain owner-only.
- Requests older than `DELAY_MAX` seconds should be rejected.

Test coverage

Cases covered

- Happy-path mint and redeem requests with event and balance checks
- Invalid token-type rejection for both mint and redeem
- Insufficient-balance rejection for both mint and redeem
- Stale-timestamp rejection for both mint and redeem
- Allowlist add/remove behavior
- Ownership transfer with upgrade-cap validation
- Migration and upgrade-cap-ID initialization checks

Cases not covered

- Zero-amount request behavior
- Acceptance of future timestamps
- Pool-account misconfiguration cases such as zero address or sender-controlled pool addresses
- Multiple requests from the same user in a single transaction
- Explicit non-owner negative tests for pool-account setters

Attack surface

- Users control `amount`, `timestamp`, `preprice`, `slippage`, and the token they present. On-chain checks only enforce allowlisting, staleness, and balance; `preprice` and `slippage` are emitted for off-chain use and are not enforced on chain.
- Funds are transferred directly to the configured pool accounts, so correct and honest pool configuration is a trust assumption.
- The module does not settle the opposite side of the trade on chain, so users rely on the off-chain fulfillment process.

6. Assessment Results

During our assessment on the scoped XAGm modules, we discovered two findings. No critical issues were found. One finding was of high impact and one was of low impact.

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.