Zellic

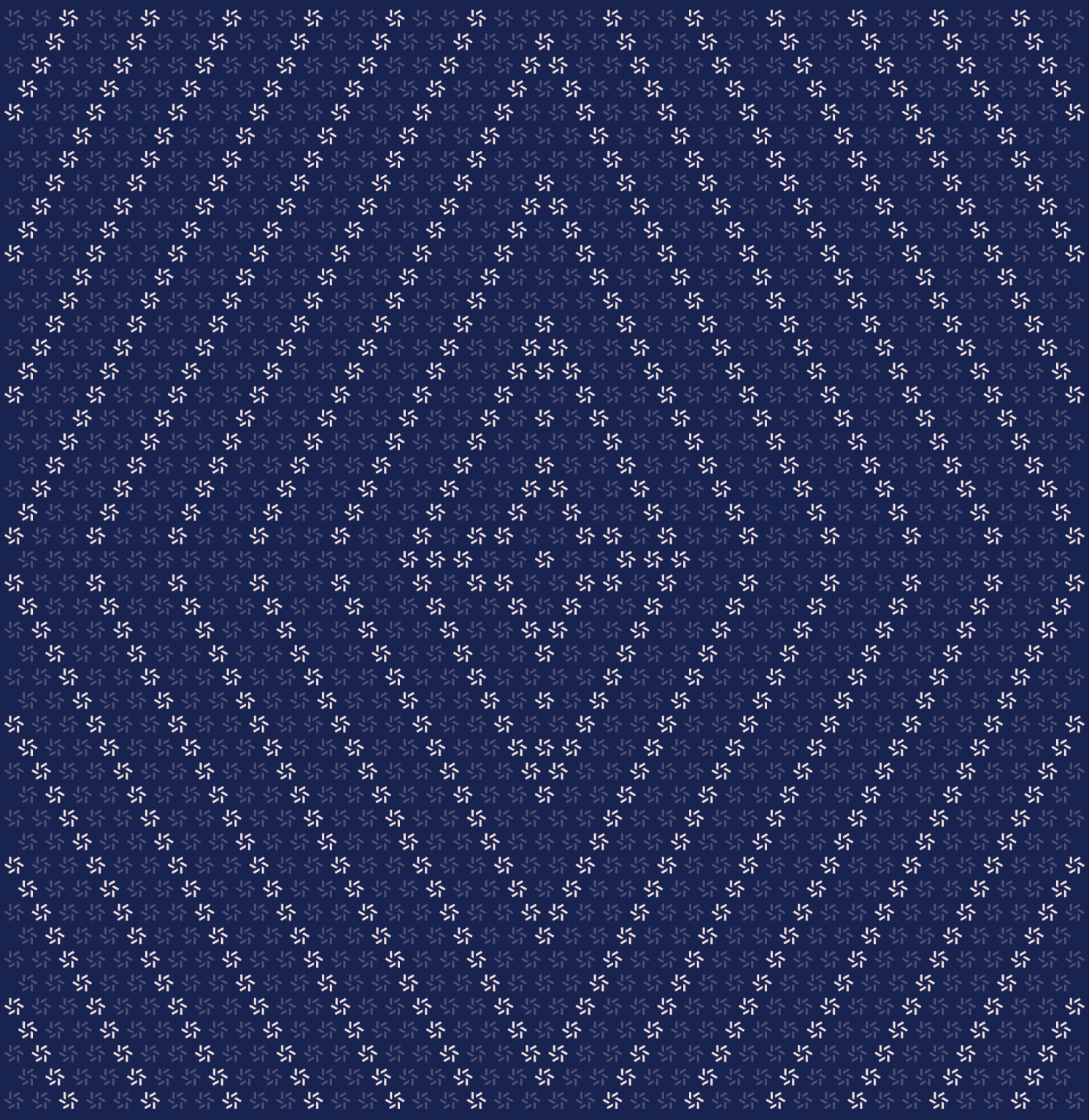
Prepared for
Brent Perreault
Jacob Torres
Paxos

Prepared by
Filipe Alves
Qingying Jie
Zellic

February 24, 2026

Paxos USDG Rewards

Smart Contract 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 reassessment for Paxos from February 9th to February 12th, 2026. This engagement is a follow-up to a prior audit of Paxos USDG Rewards, focused on reviewing the codebase after the removal of the checkpoint claims feature and additional remediation changes. During this engagement, Zellic reviewed the updated 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. This reassessment follows a prior audit of Paxos USDG Rewards, with the primary focus on verifying the correctness of the codebase after the removal of the checkpoint claims feature and reviewing additional remediation changes. We sought to answer the following questions:

- Does the removal of the checkpoint claims feature introduce any regressions or break existing reward functionality?
- Are there any ways of claiming more funds than intended? Is there any reward double counting or double claiming?
- Are there any incorrect calculations in the shares-based reward model?
- Are access controls implemented effectively to prevent unauthorized operations?
- Are there any vulnerabilities that could result in the loss of user funds?
- Are there any race conditions or ordering dependencies that could lead to potential loss of funds or incorrect reward distribution?
- Do the additional code changes introduced after the original audit's remediations introduce any new vulnerabilities?

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

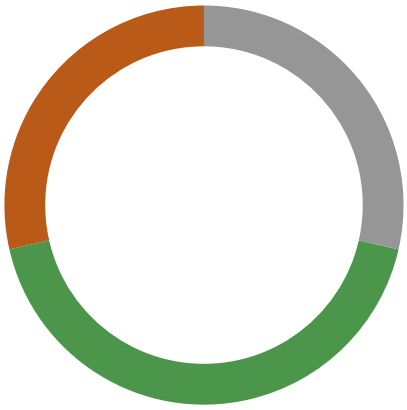
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. This engagement is a reassessment that builds on the findings and analysis of the original audit. While the entire in-scope codebase was reviewed, emphasis was placed on the changes introduced since the original audit's remediations.

1.4. Results

During our assessment on the scoped Paxos USDG Rewards contracts, we discovered seven findings. No critical issues were found. Two findings were of high impact, three were of low impact, and the remaining findings were informational in nature.

Breakdown of Finding Impacts

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



2. Introduction

2.1. About Paxos USDG Rewards

Paxos contributed the following description of Paxos USDG Rewards:

> On-chain claimable rewards system — A gas-efficient rewards system for Global Dollar Network partners that replaces 30-day delayed reconciliation with on-chain reward calculation. It uses a shares-based model where rewards compound at a configured period, enabling O(1) gas complexity for claims regardless of group size. The system supports multiple multipliers per jurisdiction, requires a funded claim source, and ensures compliance by allowing partners (not end-users) to earn and claim rewards for marketing and incentive programs.

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:

Paxos USDG Rewards Contracts

Type	Solidity
Platform	EVM-compatible
Target	paxos-token-contract-internal
Repository	https://github.com/paxosglobal/paxos-token-contract-internal ↗
Version	`Only changes between 3ce0a75...9f9618f`

Programs	`BaseStorageV3.sol`
	`ClaimableRewardsBase.sol`
	`ClaimableRewardsErrors.sol`
	`ClaimableRewardsEvents.sol`
	`ClaimableRewardsStorageV3.sol`
	`PaxosTokenClaimableRewards.sol`
	`TokenAdminEvents.sol`
	`facets/ClaimableRewardsFacet.sol`
	`facets/MultiplierMgmtFacet.sol`
	`facets/PayoutGroupFacet.sol`
	`facets/TokenAdminFacet.sol`
	`facets/TokenExtensionsFacet.sol`
	`lib/PaxosBaseAbstract.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.2 person-weeks. 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:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filipe Alves
Engineer
filipe@zellic.io

Qingying Jie
Engineer
qingying@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 9, 2026	Start of primary review period
February 11, 2026	Kick-off call
February 12, 2026	End of primary review period

3. Detailed Findings

3.1. Missing shares recalculation in multiplier update allows double claiming

Target	PayoutGroupFacet		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The `adminSetPayoutGroupMultiplier()` function forces a reward claim via `_executeClaimAll()` before changing the multiplier but fails to recalculate the group shares with the new multiplier value. After `_executeClaimAll()` resets shares using the old multiplier, only `lastClaimAllMultiplier` is updated to the new value, leaving the group shares stale.

```
// contracts/facets/PayoutGroupFacet.sol#L-129
function adminSetPayoutGroupMultiplier(uint32 payoutGroupId,
    uint32 multiplierId) external onlyPayoutGroupRegistrarRole {
    // [...]

    // Force claim before changing multiplier
    address destination = _getPayoutDestination(payoutGroupId);
    _executeClaimAll(payoutGroupId, destination);

    // [...]

    // Update payout group to new multiplier
    payoutGroup.multiplierId = uint16(multiplierId);

    uint256 newMultiplierValue = _getActiveMultiplier(multiplierId);
    payoutGroup.lastClaimAllMultiplier = StorageLib.toUint48Multiplier(
        newMultiplierValue);

    // [...]
}
```

Inside `_executeClaimAll()`, the shares are reset using the current (old) multiplier:

```
// contracts/ClaimableRewardsBase.sol#L-397
function _executeClaimAll(uint32 payoutGroupId, address destination)
    internal returns (uint256) {
    PayoutGroupData memory payoutGroup = payoutData[payoutGroupId];
```

```
        uint32 multiplierId = uint32(payoutGroup.multiplierId);

        uint256 currentMultiplier = _getActiveMultiplier(multiplierId);
        uint256 groupBalance = uint256(payoutGroup.balance);

        // [...]

        payoutGroup.lastClaimAllPeriodNum = _getCurrentPeriodNum();
        payoutGroup.lastClaimAllMultiplier
        = StorageLib.toUint48Multiplier(currentMultiplier);


        uint256 newGroupShares = SharesLib.calcShares(groupBalance, currentMultipli
            er);
    payoutGroup.shares = StorageLib.toUint64Shares(newGroupShares);

        payoutData[payoutGroupId] = payoutGroup;
        // [...]
    }
```

Since `_executeClaimAll()` operates on a memory copy and writes it back to storage, when `adminSetPayoutGroupMultiplier()` subsequently updates `lastClaimAllMultiplier` via its storage pointer, the group shares remain calculated with the old multiplier.

Consider the following scenario with two multipliers: multiplier A with a `perPeriodRate` of 0.2, and multiplier B with a `perPeriodRate` of 0.5. A payout group has an initial balance of 100 and is using multiplier A.

At period 2, the admin changes the group's multiplier from A to B. At this point, multiplier A's compounded value is 1.44 (1.2^2) and multiplier B's compounded value is 2.25 (1.5^2). During `adminSetPayoutGroupMultiplier()`, the following occurs:

1. First, `_executeClaimAll()` claims any pending rewards and resets the group shares using multiplier A: `groupShares = 100 / 1.44 = 69`.
2. Then, `lastClaimAllMultiplier` is updated to multiplier B's value: `2.25`.
3. Group shares remain at 69 (not recalculated with the new multiplier).

In the same period, performing `claimAll` again yields unearned rewards: `69 * 2.25 - 100 = 55`.

If shares had been recalculated with the new multiplier, no additional rewards would be possible: `groupShares = 100 / 2.25 = 44` and `44 * 2.25 - 100 = 0`.

Impact

A payout group can claim additional unearned rewards each time its multiplier is changed to a higher value. The extra rewards are proportional to the ratio between the new and old multiplier values, directly draining funds from the claim source.

Recommendations

Recalculate the group shares using the new multiplier value after updating `lastClaimAllMultiplier` in `adminSetPayoutGroupMultiplier()`.

Remediation

This issue has been acknowledged by Paxos, and a fix was implemented in commit `8a5fc784` ↗.

3.2. Stale period numbers allow reward double claims after maturity-period change

Target	MultiplierMgmtFacet		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The `SharesLib.handleClaimAllDetection()` function uses period-number comparison to determine whether a wallet's shares need recalculation after a `claimAll` event. It assumes period numbers are monotonically increasing — however, this assumption breaks when the maturity period or reference time is updated via `MultiplierMgmtFacet.setMaturityPeriod()` or `MultiplierMgmtFacet.setReferenceTime()`.

```
// contracts/lib/SharesLib.sol#L-138
function handleClaimAllDetection(
    uint256 balance,
    uint256 currentShares,
    uint32 walletPeriodNum,
    uint32 lastClaimAllPeriodNum,
    uint256 lastClaimAllMultiplier
) internal pure returns (uint256 adjustedShares) {
    // Check if wallet was updated AFTER the last claimAll using period numbers
    // Period numbers are monotonically increasing, so this comparison is
    always reliable
   if (walletPeriodNum > lastClaimAllPeriodNum) {
        // Wallet updated after claimAll - shares already correct
       return currentShares;
    }

    // Wallet updated before or at claimAll - claimAll zeroed rewards
    // Recalculate shares based purely on balance (zero rewards)
    return calcShares(balance, lastClaimAllMultiplier);
}
```

Period numbers are derived from the maturity period and reference time:

```
// contracts/lib/SharesLib.sol#L-69
function getCurrentPeriodNum(
```

```
    uint40 referenceTime,
    uint32 maturityPeriod,
    uint256 currentTime
) internal pure returns (uint32 periodNum) {
    // [...]
    uint256 timeElapsed = currentTime > referenceTime
        ? currentTime - referenceTime
        : 0;
    uint256 periodsElapsed = timeElapsed / maturityPeriod;
    // [...]
    return uint32(periodsElapsed);
}
```

Increasing `maturityPeriod` or moving `referenceTime` forward causes the current period number to decrease. This allows stale `walletPeriodNum` values (recorded before the update) to appear greater than newly recorded `lastClaimAllPeriodNum` values. When this occurs, the detection logic incorrectly concludes that the wallet was updated after the most recent `claimAll`, returning cached shares instead of recalculating them — enabling a user to claim rewards that were already distributed.

Consider the following scenario:

1. At period 3, a user transfers tokens, setting `walletPeriodNum = 3`. The group then executes `claimAll`, setting `lastClaimAllPeriodNum = 3`.
2. An admin doubles the maturity period, causing the current period to drop to 1.
3. At period 2, the group executes `claimAll` again, setting `lastClaimAllPeriodNum = 2`.
4. The user performs an individual claim. Since `walletPeriodNum (3) > lastClaimAllPeriodNum (2)`, the function returns the cached shares instead of recalculating, allowing the user to claim rewards that were already distributed in step 3.

Impact

When an admin increases the maturity period or moves the reference time forward, users whose `walletPeriodNum` was recorded before the update can double claim rewards. The stale period number causes the detection logic to skip recalculation, treating already claimed rewards as still pending.

Recommendations

Consider replacing period-number–based ordering with a timestamp-based approach (e.g., `block.timestamp`) for tracking the relative ordering of wallet updates and `claimAll` events, as timestamps are guaranteed to be monotonically increasing regardless of configuration changes.

Remediation

This issue has been acknowledged by Paxos, and a fix was implemented in commit `0af60714` ↗.

3.3. Missing upper bound in `cancelPermits` allows permit self-bricking

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

Description

The `cancelPermits()` function allows a user to increment their nonce by an arbitrary amount with no upper bound. A user could accidentally or intentionally pass an excessively large `count` value, effectively exhausting their nonce space and permanently disabling all future permit authorizations for their account.

```
// contracts/facets/TokenExtensionsFacet.sol#L-107
function cancelPermits(uint256 count) external {
    if (count == 0) revert InvalidNonceCount();

    _nonces[msg.sender] += count;
    emit PermitInvalidated(msg.sender, _nonces[msg.sender]);
}
```

Since `_nonces` is a `uint256` mapping and there is no upper bound check on `count`, a user could pass `type(uint256).max` to jump the nonce to a value from which no further valid permits can be signed.

Impact

A user can permanently disable their own ability to use EIP-2612 permits by passing an excessively large `count` value. While this is a self-inflicted issue and does not affect other users, it is irreversible and could result from a front-end bug or user mistake.

Recommendations

Add a reasonable upper bound on the `count` parameter to prevent accidental nonce exhaustion.

Remediation

This issue has been acknowledged by Paxos, and a fix was implemented in commit `ff44b973` ↗.

3.4. Frozen-address validation bypassed on indirect reward claims

Target	ClaimableRewardsBase		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The external claim functions in ClaimableRewardsFacet properly validate destinations via `_validateNotFrozen()`:

```
// contracts/ClaimableRewardsBase.sol#L-151
function _validateNotFrozen(address caller, address account,
    address destination) internal view {
    if (_isFrozen(caller)) revert AddressFrozen();
    if (account != address(0) && _isFrozen(account)) revert AddressFrozen();
    if (_isFrozen(destination)) revert AddressFrozen();
}
```

However, claims triggered indirectly through privileged operations bypass this check, as they call `_executeClaimAll()` or `_claimIndividualRewards()` directly without validating the destination:

- `deletePayoutGroup()` calls `_executeClaimAll()` with the configured destination
- `adminSetPayoutGroupMultiplier()` calls `_executeClaimAll()` with the configured destination
- `registrarUnregisterRewardAddress()`, `registrarUnregisterRewardAddressBatch()`, and `unregisterRewardAddress()` call `_claimIndividualRewards()` without frozen checks

Additionally, `adminSetPayoutGroupDestination()` and `setPayoutGroupDestination()` allow setting a frozen address as the destination without validation, which can then be used by the above indirect paths.

Impact

The impact is low since these operations require privileged roles (`PAYOUT_GROUP_REGISTRAR_ROLE` or `PAYOUT_ADMIN_ROLE`). However, it creates inconsistent behavior where direct claims to a frozen destination revert while indirect claims through privileged operations succeed.

Recommendations

Consider adding frozen-address checks to the indirect claim paths.

Remediation

This issue has been acknowledged by Paxos, and a fix was implemented in commit 164896bb ↗.

3.5. Missing pause enforcement on admin-triggered reward claims

Target	ClaimableRewardsBase		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The contract can prevent users from transferring tokens by setting `globalTransferSettings.paused` to `true`. However, certain code paths bypass this global pause to claim rewards and execute token transfers:

- `deletePayoutGroup()` calls `_executeClaimAll()`, which transfers rewards from the claim source
- `adminSetPayoutGroupMultiplier()` calls `_executeClaimAll()` before changing the multiplier
- `registrarUnregisterRewardAddress()`, `registrarUnregisterRewardAddressBatch()`, and `unregisterRewardAddress()` call `_claimIndividualRewards()`, which transfers rewards

Most of these functions are restricted to `PAYOUT_GROUP_REGISTRAR_ROLE`, meaning they can only be called by privileged accounts. However, `unregisterRewardAddress()` is callable by the payout group's claimer or manager — external parties who can independently trigger reward transfers while the contract is paused.

Impact

The impact is low because these paths cannot be exploited by arbitrary users. However, this creates a gap in the pause mechanism where privileged operations can move tokens despite the contract being paused, which may violate the intended semantics of the global pause.

Recommendations

Add a pause check to `unregisterRewardAddress()` to enforce the global pause on the externally accessible path. For admin-only functions, confirm whether allowing token movement during a pause is intentional. If the pause is meant to halt all token transfers regardless of the caller's role, consider adding pause checks to those paths as well.

Remediation

This issue has been acknowledged by Paxos, and a fix was implemented in commit 5199f505 ↗.

3.6. Redundant `DOMAIN_SEPARATOR_DEPRECATED` storage slot from EIP712Domain inheritance

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

Description

The contract ClaimableRewardsBase inherits from both BaseStorageV3 and EIP712Domain contracts, each of which declares a private `DOMAIN_SEPARATOR_DEPRECATED` storage variable:

```
// contracts/BaseStorageV3.sol#L-120
bytes32 private DOMAIN_SEPARATOR_DEPRECATED; // solhint-disable-line
    var-name-mixedcase
```

```
// contracts/lib/EIP712Domain.sol#L-18
bytes32 private DOMAIN_SEPARATOR_DEPRECATED; // solhint-disable-line
    var-name-mixedcase
```

Since private variables are scoped to their declaring contract, inheriting both results in two separate storage slots for `DOMAIN_SEPARATOR_DEPRECATED`. The slot from EIP712Domain is redundant.

Impact

The redundant storage slot introduces unnecessary storage overhead and may cause confusion during future maintenance or storage layout analysis. It does not affect functionality.

Recommendations

Remove the EIP712Domain inheritance from ClaimableRewardsBase if it is only used for the `EIP712_VERSION_PREFIX` constant. If the inheritance is required for compatibility, document the storage layout overlap.

Remediation

This issue has been acknowledged by Paxos, and a fix was implemented in commit `1250ca31` ↗.

3.7. Missing pause check in the function `cancelPermits` creates inconsistency with permit flow

Target	TokenExtensionsFacet		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The function `cancelPermits` in the contract TokenExtensionsFacet does not validate the global pause state, making it inconsistent with the function `_permit`, which explicitly reverts with the error `ContractPaused` when the contract is paused.

The `_permit` function correctly gates execution behind the pause check:

```
// contracts/facets/TokenExtensionsFacet.sol#L-410
function _permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    bytes memory signature
) internal {
    if (globalTransferSettings.paused) revert ContractPaused();
    // [...]
}
```

But the function `cancelPermits` has no such check:

```
// contracts/facets/TokenExtensionsFacet.sol#L-107
function cancelPermits(uint256 count) external {
    if (count == 0) revert InvalidNonceCount();

    _nonces[msg.sender] += count;
    emit PermitInvalidated(msg.sender, _nonces[msg.sender]);
}
```

This means that while users cannot create new permit approvals during a pause, they can cancel and invalidate existing ones. The practical security impact is limited since token transfers are still subject to the global pause state, including through the function `transferWithAuthorization` or

function `receiveWithAuthorization`. However, the inconsistency is notable because the function `_cancelAuthorization` (the EIP-3009 counterpart) does enforce the pause state:

```
// contracts/facets/TokenExtensionsFacet.sol#L-441
function _cancelAuthorization(
    address authorizer,
    bytes32 nonce,
    bytes memory signature
) internal {
   if (globalTransferSettings.paused) revert ContractPaused();
    // [...]
}
```

This makes the behavior across the permit and authorization flows uneven.

Impact

The inconsistency does not pose a direct security risk, as token movement remains blocked during a pause. However, it creates an uneven enforcement of the pause semantics across related flows.

Recommendations

Add a pause check to the function `cancelPermits` for consistency with the rest of the paused-state enforcement.

Remediation

This issue has been acknowledged by Paxos, and a fix was implemented in commit bdedc625 ↗.

4. 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.

The following will focus on the changes made from commit `3ce0a753` to `9f9618fc`.

4.1. Contract: PaxosTokenClaimableRewards

The contract PaxosTokenClaimableRewards is the main contract, which implements core ERC-20 functions (`transfer`, `approve`, `balanceOf`) and minting/burning, and it routes unknown selectors to facets with different functionalities via `delegatecall`.

There was one system-wide notable change: the removal of the checkpoint period. This means

- the calculation of claimable rewards now depends only on the maturity period and no longer relies on the checkpoint period,
- the maturity period number is now used to determine the freshness of the shares recorded in a wallet compared to the `lastClaimAllPeriodNum` in the payout group, and
- the struct `TokenAccountData` no longer includes the `cachedMultiplier`, `cachedCheckpointNum`, and `cachedCheckpointAvailableRewards` fields. A `periodNum` field has been added.

The following outlines system-wide minor changes:

- The `DOMAIN_SEPARATOR` is recalculated on every access rather than read from a state variable.
- The contract ClaimableRewardsBase newly inherits from the contract EIP712Domain. The contract EIP712Domain contains a constant `EIP712_VERSION_PREFIX`, and an extra state variable `DOMAIN_SEPARATOR_DEPRECATED` (which has the same name as a state variable defined in another inherited contract, BaseStorageV3). The constant `EIP712_VERSION_PREFIX` will be used when calculating the hash of EIP-712 typed data.

4.2. Facet: Payout group

Payout groups are groups of addresses that accumulate rewards together. The contract PayoutGroupFacet supports operations such as creating, deleting, and modifying configurations of payout groups as well as registering or unregistering addresses to/from payout groups.

The contract PayoutGroupFacet has been updated primarily to reflect the removal of the checkpoint period, with the following corresponding code changes:

- The struct `PayoutGroupData` no longer includes the `cachedCheckpointNum` and `cachedCheckpointAvailableRewards` fields. Instead, a `lastClaimAllPeriodNum` field

has been added.
- Calls to the `_executeClaimAll` and `_setBalanceData` functions no longer require checkpoint-related parameters.
- Unregistering an account from a payout group no longer requires updating the payout group's checkpoint rewards.
- The functions `getAccountCheckpointRewards`, `getAccountCheckpointNum`, `getPayoutGroupCheckpointRewards`, and `getPayoutGroupCheckpointNum` have been removed.

The following outlines other changes:

- The `proposeRegisterRewardAddress` and `acceptRegisterRewardAddress` functions now allow orphaned accounts whose payout group has been deleted to register.
- A new function, `availableRewardsOfBatch`, has been introduced to support batch queries of available rewards for multiple accounts.

4.3. Facet: Multiplier management

A multiplier defines how yield accrues over time. Starting from the configured reference time, every multiplier grows at each maturity period according to the following formula:

```
multiplier = multiplier x (1 + perPeriodRate)
```

The contract MultiplierMgmtFacet has been updated primarily to reflect the removal of the checkpoint period, with the following corresponding code changes:

- The struct `MultiplierData` no longer includes the `cachedCheckpointMultiplier` and `cachedCheckpointNum` fields.
- The `setCheckpointPeriod`, `getCheckpointPeriod`, and `_updateCheckpointCache` functions have been removed.
- The length of the maturity period no longer needs to be constrained by a multiple relationship with the checkpoint period.

The following outlines other changes:

- When setting the claim source, additional checks have been added to ensure the address is not the zero address and that the address is not frozen.
- In the `getNextAPR`, `getNextAPY`, and `getSwitchTime` functions, if the queried `multiplierId` does not exist, the behavior has changed from returning zero to reverting with the `MultiplierIndexNotFound` error.
- During initialization of the contract PaxosTokenClaimableRewards, `minRate` and `maxRate` must now be set. As a result, the `_validateAPRBounds` function no longer determines whether to enforce bounds based on `minRate` or `maxRate` being zero; instead, it requires the parameter `apr` to always fall between `minRate` and `maxRate`.

4.4. Facet: Claimable rewards

The contract ClaimableRewardsFacet provides external functions with different claim methods for the active payout group.

The following outlines the changes made in this contract:

- All claim methods no longer support claiming up to the latest checkpoint and only allow claims up to the current time.
- All claim methods have been updated to return a value representing the total amount of rewards claimed.
- The function `_processBatchClaims` emits an event `RewardsClaimedBatch` after processing all the individual claims.

4.5. Facet: Token admin

The contract TokenAdminFacet provides privileged accounts with functions to manage the contract and user addresses.

A notable change in this contract is the introduction of the reward freeze/unfreeze functionality:

- Freezing an account will remove it from the current payout group, which halts its reward accrual. The mapping `frozenData` is used to record the account's group and current reward amount.
- Upon unfreezing, if the original payout group still exists, the account's rewards at the time of freezing are restored. Otherwise, its frozen reward will be lost.

As a result, when wiping the balance of a frozen account, there is no need to update the payout group data.

5. Assessment Results

During our assessment on the scoped Paxos USDG Rewards contracts, we discovered seven findings. No critical issues were found. Two findings were of high impact, three were of low impact, and the remaining findings were informational in nature.

The codebase is well-structured and demonstrates a mature engineering approach, with clear separation of concerns across facets and libraries. The removal of the checkpoint claims feature was executed cleanly, resulting in a simpler and more maintainable reward system.

However, this reassessment uncovered two high-severity findings related to the shares-based reward model that could allow double claiming of rewards. We recommend that Paxos prioritize remediating these issues before deployment. Once the high-severity findings are addressed, consider deploying the contracts to testnet for several weeks to validate intended behavior under real conditions. This would be beneficial before mainnet deployment.

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.