// Security Assessment

10.17.2025 - 11.03.2025

Paxos Token Contracts

Paxos

Submit Feedback

HALBORN

Paxos Token Contracts - Paxos

Prepared by: HALBORN

Last Updated 11/21/2025

Date of Engagement: October 17th, 2025 - November 3rd, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
8	0	1	3	3	1

TABLE OF CONTENTS

1. INTRODUCTION

`Paxos` engaged Halborn to conduct a security assessment on their smart contracts beginning on October 17th, 2025 and ending on November 3rd, 2025. The security assessment was scoped to the smart contracts provided in the `Paxos Github repository`, provided to the Halborn team. Commit hash and further details can be found in the Scope section of this report.

The reviewed contracts manage the lifecycle of rewards accrual and claiming, payout group coordination, multiplier-linked yield adjustments, and administrative operations such as pausing, freezing, and upgrading of the Paxos USDG token.

2. ASSESSMENT SUMMARY

`Halborn` was provided with 11 days for this engagement and assigned a full-time security engineer to assess the security of the smart contracts in scope. The assigned engineer possess deep expertise in blockchain and smart contract security, including hands-on experience with multiple blockchain protocols.

The objective of this assessment is to:

- Identify potential security issues within the `Paxos` protocol smart contracts.
- Ensure that smart contract of `Paxos` protocol functions operate as intended.

In summary, `Halborn` identified several areas for improvement to reduce the likelihood and impact of security risks, which were mostly addressed by the `Paxos team`. The main ones were the following:

- `Preventing self-transfers that would inflate balances and corrupt shares.`
- `Moving multiplier-level balance and share aggregates entirely off-chain, preventing incorrect rewards scaling.`
- `Updating the logic to prevent a reward source from being registered into a payout group, and moving rewards calculations off-chain to ensure consistent rewards handling.`
- `Updating payout group and multiplier when admin reclaims.`

3. TEST APPROACH AND METHODOLOGY

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture and purpose of the `Paxos` protocol.
- Manual code review and walkthrough of the `Paxos` in-scope contracts.
- Manual assessment of critical `Solidity` variables and functions to identify potential vulnerability classes.
- Manual testing using custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`).
- Local deployment and testing with (`Foundry`, `Remix IDE`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: paxos-token-contract-internal

(b) Assessed Commit ID: 769777f

(c) Items in scope:

- contracts/BaseStorageV3.sol
- contracts/ClaimableRewardsBase.sol
- contracts/ClaimableRewardsErrors.sol
- contracts/ ClaimableRewardsEvents.sol
- contracts/ ClaimableRewardsStorageV3.sol
- contracts/PaxosTokenClaimableRewards.sol
- contracts/ TokenAdminEvents.sol
- contracts/facets/ClaimableRewardsFacet.sol
- contracts/facets/MultiplierMgmtFacet.sol
- contracts/facets/PayoutGroupFacet.sol
- contracts/facets/TokenAdminFacet.sol
- contracts/facets/ TokenExtensionsFacet.sol
- contracts/lib/fixture/RateLimitFixture.sol
- contracts/lib/EIP2612.sol
- contracts/lib/EIP2612Definitions.sol
- contracts/lib/EIP3009Definitions.sol
- contracts/lib/Roles.sol
- contracts/lib/EIP2612Definitions.sol
- contracts/lib/MultiplierGrowthLib.sol
- contracts/lib/SharesLib.sol
- contracts/lib/EIP3009.sol
- contracts/lib/PaxosBaseAbstract.sol
- contracts/lib/StorageLib.sol
- contracts/stablecoins/USDG.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- c630207
- dd3792d
- 68dbb31

- 48cfc16
- 7b714db
- 6155c11

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	3	3	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
SELF-TRANSFER INFLATES BALANCE AND CORRUPTS SHARES	HIGH	SOLVED - 11/06/2025
BALANCE/SHARE AGGREGATES ARE NOT MIGRATED WHEN SWITCHING A GROUP TO A NEW MULTIPLIER,LEADING TO INCORRECT REWARD SCALING	MEDIUM	SOLVED - 11/07/2025
REWARD SOURCE STARTS EARNING ON FULL BALANCE AFTER FIRST REWARD TRANSFER	MEDIUM	SOLVED - 11/07/2025
ADMIN RECLAIM IGNORES PAYOUT GROUP AND MULTIPLIER UPDATES	MEDIUM	SOLVED - 11/07/2025
FROZEN ACCOUNTS CAN BE REGISTERED INTO PAYOUT GROUPS	LOW	SOLVED - 11/07/2025
NO MECHANISM TO INVALIDATE PENDING EIP-2612 PERMITS BEFORE EXPIRY	LOW	SOLVED - 11/06/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF BALANCE MIGRATION WHEN CHANGING CLAIM SOURCE MAY CAUSE REWARD CLAIM FAILURES	LOW	RISK ACCEPTED - 11/07/2025
PUBLIC INITIALIZER MAY ALLOW UNAUTHORIZED INITIALIZATION IF LEFT UNINITIALIZED	INFORMATIONAL	ACKNOWLEDGED - 11/07/2025

7. FINDINGS & TECH DETAILS

7.1 SELF-TRANSFER INFLATES BALANCE AND CORRUPTS SHARES

// HIGH

Description

In `ClaimableRewardsBase._transfer(from, to, value)`, the balances for the involved accounts are first fetched; subsequent balance updates are performed as shown:

```
819 function _transfer(address from, address to, uint256 value) internal virtual override {
820     // Check for zero address - use inherited error from PaxosBaseAbstract
821     if (to == address(0)) revert ZeroAddress();
822     if (_isFrozen(to) || _isFrozen(from)) revert AddressFrozen();
823
824     // Early exit for zero value transfers
825     if (value == 0) {
826         emit Transfer(from, to, value);
827         return;
828     }
829
830     // Single SLOAD for both wallets (2 SLOAD total)
831     TokenAccountData memory fromWallet = _getBalanceData(from);
832     TokenAccountData memory toWallet = _getBalanceData(to);
833
834     if (value > fromWallet.balance) revert InsufficientFunds();
835
836     // Calculate new balances
837     uint256 newFromBalance;
838     uint256 newToBalance;
839     unchecked {
840         newFromBalance = uint256(fromWallet.balance) - value;
841         newToBalance = uint256(toWallet.balance) + value;
842     }
843     //...
844 }
```

When `from == to`, the same balance is updated twice. As a result, arbitrary token creation is enabled by repeatedly transferring to the same address.

Proof of Concept

Insert this test in the `GlobalMetricsTest.js` test file:

```
describe('Global Metrics Tests', function () {
//...
 it.only('self transfer acts as mint', async function () {
        const mintAmount = ethers.parseUnits("10", 12);
        const numberOfTransfers = BigInt(5);

        // Initial mint: Mint `mintAmount` tokens to acc address
```

```
        await this.token.connect(this.owner).mint(this.acc.address, mintAmount);

        // Self-transfer loop:
        // Each transfer to self effectively mints additional tokens
        // After `numberOfTransfers` iterations, balance increases by that many times
        for (let i = 0; i < Number(numberOfTransfers); i++) {
        await this.token.connect(this.acc).transfer(this.acc.address, ethers.parseUnits("10", 12));
        }

        // Expected balance:
        // initial minted amount + numberOfTransfers times self-transfer mint
        expect(await this.token.balanceOf(this.acc)).to.equal(numberOfTransfers * mintAmount + mintAmoun

    });
}
```

The test passes, indicating that self-transfers are allowed and incorrectly behave like a minting operation:

Copy Code

```
$ npx hardhat test

  Global Metrics Tests
    ✔ self transfer acts as mint

    1 passing (2s)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (7.5)

Recommendation

It must be verified that the condition `sender != receiver` is enforced in the `ClaimableRewardsBase._transfer()` function.

Remediation Comment

SOLVED: The **Paxos team** solved the issue in the specified commit id by returning the `_transfer()` early when the sender is the receiver.

Remediation Hash

https://github.com/paxosglobal/paxos-token-contract-internal/commit/c630207da4dcdbb9a1d1eddbbfabe17d91141735

7.2 BALANCE/SHARE AGGREGATES ARE NOT MIGRATED WHEN SWITCHING A GROUP TO A NEW MULTIPLIER,LEADING TO INCORRECT REWARD SCALING

// MEDIUM

Description

The `PayoutGroupFacet.adminSetPayoutGroupMultiplier()` function permits an admin to move a payout group from one multiplier to another. The implementation updates the group's `multiplierId` and adjusts the `payoutGroupCount` for the involved multipliers, but the group's balances and shares are not migrated from the previous multiplier aggregate to the new one.

Copy Code

```
141 function adminSetPayoutGroupMultiplier(uint32 payoutGroupId, uint32 multiplierId) external onlyPa
142     //...
143
144     // Force claim before changing multiplier
145     address claimer = payoutIdToClaimer[payoutGroupId];
146     address destination = _getPayoutDestination(uint32(payoutGroupId));
147     _executeClaimAll(claimer, destination);
148
149     // Update payout group counts for both multipliers
150     multiplierBals[oldMultId].payoutGroupCount--;  // Decrement old multiplier
151     multiplierBals[multiplierId].payoutGroupCount++;     // Increment new multiplier
152
153     // Update payout group to new multiplier (shares model)
154     uint256 newMultiplier = _getActiveMultiplier(multiplierId); // Already at 12 decimals
155     payoutGroup.lastClaimAllBaseMultiplier = StorageLib.toUint56Multiplier(newMultiplier);
156     payoutGroup.multiplierId = uint24(multiplierId); // uint24 in V3
157
158     emit PayoutGroupMultiplierUpdated(claimer, oldMultId, multiplierId);
159 }
```

Prior to the switch, the group's balances and shares were included in the aggregate totals under `multiplierBals[oldMultId]`. After the switch, the group's logical association is updated to the new multiplier, while its previous contributions remain recorded under the old multiplier aggregate. Although the new multiplier's `payoutGroupCount` is incremented, the group's balances and shares are not transferred.

This discrepancy creates a mismatch between `payoutGroup.multiplierId` and the actual aggregated state within `multiplierBals`. Over time, the mismatch is propagated into reward calculations, because rewards are computed from the per-multiplier aggregated balances and shares.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (5.0)

Recommendation

When a payout group's multiplier is changed, its balance and shares must be explicitly migrated from the old multiplier to the new multiplier by updating the corresponding records.

Remediation Comment

SOLVED: The **Paxos team** solved the issue in the specified commit id by removing multiplier-level aggregates entirely, moving compounding calculations off-chain to eliminate migration errors and improve gas efficiency.

Remediation Hash

https://github.com/paxosglobal/paxos-token-contract-internal/commit/dd3792d2a53f3ffa2100b2b0cc4f3926a46635d5

7.3 REWARD SOURCE STARTS EARNING ON FULL BALANCE AFTER FIRST REWARD TRANSFER

// MEDIUM

Description

When a reward source account is first registered to a payout group in `_internalRegisterRewardAddress()`, it computes an effective balance by subtracting the total available rewards from all multipliers, and then mints shares based on that reduced balance. This is clearly meant to enforce **the invariant that the reward source should not earn rewards on the rewards that belong to others** invariant.

```
function _internalRegisterRewardAddress(uint32 payoutGroupId, address account) internal {
        //...

        // SPECIAL CASE: Claim source earns on effective balance (actual balance - others' reward
        if (_isRewardSource(account)) {
            // Get total available rewards across all multipliers (excluding this account since n
            // Direct internal call - no external callback needed
            uint256 totalOthersRewards = _getTotalAvailableRewardsInternal();

            // Calculate effective balance for claim source
            uint256 effectiveBalance = uint256(wallet.balance) >= totalOthersRewards
                ? uint256(wallet.balance) - totalOthersRewards
                : 0;

            balanceForShares = effectiveBalance;
            newShares = SharesLib.calcShares(effectiveBalance, currentMultiplier);
        } else {
            // NORMAL CASE: Regular accounts use actual balance
            balanceForShares = uint256(wallet.balance);
            newShares = SharesLib.calcShares(uint256(wallet.balance), currentMultiplier);
        }

        // Update account with new payout group (shares model)
        _setBalanceData(account, uint256(wallet.balance), newShares, currentPeriodNum, uint256(pa

        //...
    }
```

However, in later updates in `_updateBalanceForRewardTransfer()`, the same account is treated like a normal account: its shares are recomputed from the full `newBalance` using `SharesLib.calcShares(newBalance, currentMultiplier)` without reapplying the **effective balance** rule. This means the same reward source account can start earning shares on the whole balance after the first reward transfer, breaking the earlier invariant and creating inconsistent share bases for the same address, which leads the reward source to accumulate more shares than intended, which affects multiplier-level and payout-group-level accounting.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:M (5.0)

Recommendation

In `updateBalanceForRewardTransfer()` (and in any other balance-update path that touches registered accounts), reward-source accounts should be detected using the same logic as in `_internalRegisterRewardAddress()`, and shares should be recomputed using the effective balance (balance – totalAvailableRewards) instead of the raw balance. Alternatively, registration of claim-source accounts to payout groups should be prohibited so that this inconsistent path cannot occur.

Remediation Comment

SOLVED: The **Paxos team** solved the issue in the specified commit id by removing multiplier-level tracking and moving reward calculations off-chain and preventing claim source from being registered into a payout group.

Remediation Hash

https://github.com/paxosglobal/paxos-token-contract-internal/commit/68dbb3142a9cd7d9995294144cdbf257cd645128

7.4 ADMIN RECLAIM IGNORES PAYOUT GROUP AND MULTIPLIER UPDATES

// MEDIUM

Description

The `TokenAdminFacet.reclaimToken()` function transfers the contract's entire token balance to the owner by directly setting the contract balance to zero and increasing the owner's balance.

Copy Code

```
150 function reclaimToken() external onlyRole(DEFAULT_ADMIN_ROLE) {
151     uint256 _balance = _getBalance(address(this));
152
153     // Get owner directly - inherited from AccessControlDefaultAdminRulesUpgradeable
154     address owner = owner();
155
156     _setBalance(address(this), 0);
157     _setBalance(owner, _getBalance(owner) + _balance);
158     emit Transfer(address(this), owner, _balance);
159 }
```

The function does not invoke the standard balance-change flow (`_processWalletChange()` / `_processWalletChangeWithCleanup()`) that is used elsewhere in the token to maintain consistency of payout-group and multiplier-level aggregates. As a result, if the owner address is registered in a payout group, the owner's wallet shares, period number, and the group/multiplier aggregate balances will not be updated to reflect the newly received tokens.

An inconsistent state is therefore created: the owner's balance is increased while the payout group and multiplier aggregates continue to reflect the previous (smaller) balance. Subsequent reward calculations for the owner or their group may be miscounted.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:M (5.0)

Recommendation

Update `reclaimToken()` to route the balance change through the same wallet-processing logic used by mint/burn/transfer.

Remediation Comment

SOLVED: The **Paxos team** solved the issue in the specified commit id by calling `_transfer()` directly instead of `_setBalance()` when admin reclaims.

Remediation Hash

https://github.com/paxosglobal/paxos-token-contract-internal/commit/48cfc16b4420b063e2120acd3fe93e51b6d8106e

7.5 FROZEN ACCOUNTS CAN BE REGISTERED INTO PAYOUT GROUPS

// LOW

Description

`PayoutGroupFacet.registrarRegisterRewardAddress()` does not verify whether the target account is frozen prior to registration into a payout group. As a result, frozen or sanctioned addresses can be onboarded, which conflicts with standard freeze semantics (freeze = exclusion from token operations and participation).

Copy Code

```
291 function registrarRegisterRewardAddress(uint32 payoutGroupId, address account) external onlyPayou
292     _internalRegisterRewardAddress(uint32(payoutGroupId), account);
293 }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:L/Y:L (3.1)

Recommendation

Registration of frozen accounts into payout groups should be disallowed.

Remediation Comment

SOLVED: The **Paxos team** solved the issue in the specified commit id by preventing frozen accounts from being registered into a payout group.

Remediation Hash

https://github.com/paxosglobal/paxos-token-contract-internal/commit/7b714db305c9d2d1cd8c973d8ab8fab71ccc36ff

7.6 NO MECHANISM TO INVALIDATE PENDING EIP-2612 PERMITS BEFORE EXPIRY

// LOW

Description

The `EIP-2612` implementation lacks a function for canceling a signed but unused permit (for example, `cancelPermit()` or equivalent). Once a permit signature is issued off-chain, it can be reused by anyone holding that signed data until it is executed or expires. This exposes users to risk if they wish to revoke pending permits without waiting for expiry.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (2.5)

Recommendation

A cancellation mechanism should be implemented, such as one equivalent to `EIP-3009.cancelAuthorization()`, or an alternative mechanism that allows users to increase the transaction nonce should be provided.

Remediation Comment

SOLVED: The **Paxos team** solved the issue in the specified commit id by introducing a new function to cancel pending permits by increasing the user nonce.

Remediation Hash

https://github.com/paxosglobal/paxos-token-contract-internal/commit/6155c119168b692bf057b3e37ec19d71914545f4

7.7 LACK OF BALANCE MIGRATION WHEN CHANGING CLAIM SOURCE MAY CAUSE REWARD CLAIM FAILURES

// LOW

Description

`MultiplierMgmtFacet.setClaimSource()` allows the admin to change the global reward source address. The old source is simply unmarked as `rewardSource`, and the new source is marked. However, the function does not transfer the reward balance from the old source to the new source, or run a funding check to ensure the new source can cover existing accrued rewards.

Because the protocol's reward accounting assumes all accrued rewards can be pulled from `adminConfig.claimSource`, switching to an empty or underfunded address can make subsequent claims revert with `InsufficientClaimSourceBalance()`, DoSing claims even though the old source still holds the tokens.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

When the claim source is changed, the net available reward balance must be transferred from the previous source to the replacement source, or the replacement source must hold at least `totalAvailableRewards()` prior to the source address being updated.

Remediation Comment

RISK ACCEPTED: The **Paxos team** accepted the risk of this finding.

7.8 PUBLIC INITIALIZER MAY ALLOW UNAUTHORIZED INITIALIZATION IF LEFT UNINITIALIZED

// INFORMATIONAL

Description

If the `USDG` contract is deployed but `initialize()` has not yet been executed, the public `initializeV3()` function (which uses the same `reinitializer(3)` modifier) could be invoked. As a consequence, all admin roles could be obtained before the legitimate initialization is performed. The issue exists because initialization of version 3 can be performed independently by both functions and no access restrictions are present on `initializeV3()`.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It must be ensured that deployments and initialization are performed within the same transaction by the deployment script.

Remediation Comment

ACKNOWLEDGED: The **Paxos team** acknowledged the risk of this finding.
