// Security Assessment

03.03.2026 - 03.03.2026

Base Contracts

Gold Token SA (MKS Pamp)

HALBORN

Base Contracts - Gold Token SA (MKS Pamp)

Prepared by: HALBORN

Last Updated 03/16/2026

Date of Engagement: March 3rd, 2026 - March 3rd, 2026

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
6	0	0	0	3	3

TABLE OF CONTENTS

1. INTRODUCTION

`Gold Token` engaged `Halborn` to perform a security assessment of their smart contracts on March 3rd, 2026. The assessment scope was limited to the smart contracts provided to Halborn. Commit hashes and additional details are available in the Scope section of this report.

The `DGLD` codebase in scope consists of smart contracts implementing an upgradeable gold-backed ERC20 token designed for seamless L1↔L2 bridging via the Optimism Standard Bridge, featuring role-based access control, pausability, address blacklisting, and a one-time post-exploit balance correction mechanism to remediate illicitly minted tokens following a zero-value transfer vulnerability on the Base network.

2. ASSESSMENT SUMMARY

`Halborn` was allocated 1 days for this engagement and assigned 1 full-time security engineers to conduct a comprehensive review of the smart contracts within scope. The engineers are experts in blockchain and smart contract security, with advanced skills in penetration testing and smart contract exploitation, as well as extensive knowledge of multiple blockchain protocols.

The objectives of this assessment are to:

- Identify potential security vulnerabilities within the smart contracts.
- Verify that the smart contract functionality operates as intended.

In summary, `Halborn` identified several areas for improvement to reduce the likelihood and impact of security risks, which were fully addressed by the `Gold Token team`. The main recommendations were:

- `Implement increaseAllowance() and decreaseAllowance().`
- `Implement a two-step transfer pattern.`
- `Introduce a privileged setter function restricted to DEFAULT_ADMIN_ROLE.`

3. TEST APPROACH AND METHODOLODY

`Halborn` conducted a combination of manual code review and automated security testing to balance efficiency, timeliness, practicality, and accuracy within the scope of this assessment. While manual testing is crucial for identifying flaws in logic, processes, and implementation, automated testing enhances coverage of smart contracts and quickly detects deviations from established security best practices.

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

- Research into the platform's architecture, purpose and use.
- Manual code review and walkthrough of smart contracts to identify any logical issues.
- Comprehensive assessment of the safety and usage of critical Solidity variables and functions within scope that could lead to arithmetic-related vulnerabilities.
- Local testing using custom scripts (`Foundry`).
- Fork testing against main networks (`Foundry`).
- Static security analysis of scoped contracts, and imported functions (`Slither`).

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
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
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
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: gtsa-base-smart-contracts

(b) Assessed Commit ID: 82cbf9c

Out-of-Scope: External dependencies and economic attacks.

FILE

(a) Submitted File: gtsa-base-smart-contracts-audit-v2.zip

(b) Items in scope:

- /gtsa-base-smart-contracts-audit-v2/src/Blacklistable.sol
- /gtsa-base-smart-contracts-audit-v2/src/DGLDToken.sol
- /gtsa-base-smart-contracts-audit-v2/src/UpgradeableOptimismMintableERC20.sol

REMEDIATION COMMIT ID:

- fb9f4c6
- a6c2afe
- 84a862a
- 24a16a5
- adbe37f
- 7ee11d6

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	3

INFORMATIONAL
3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
ERC20 APPROVAL RACE CONDITION ALLOWS A SPENDER TO EXCEED THE INTENDED ALLOWANCE	LOW	SOLVED - 03/08/2026
SINGLE-STEP ADMIN TRANSFER IN CHANGEROLESADMIN() CAN PERMANENTLY LOCK ALL PRIVILEGED ROLES	LOW	SOLVED - 03/08/2026
IMMUTABLE BRIDGE ADDRESS IN __UPGRADEABLEOPTIMISMMINTABLEERC20_INIT() PREVENTS RECOVERY FROM BRIDGE MIGRATION OR MISCONFIGURATION	LOW	SOLVED - 03/08/2026
MISSING ZERO ADDRESS VALIDATION FOR _BRIDGE AND _REMOTETOKEN AND MISSING ZERO AMOUNT VALIDATION IN MINT() AND BURN()	INFORMATIONAL	SOLVED - 03/08/2026
REDUNDANT BLACKLIST AND UNBLACKLIST OPERATIONS EMIT MISLEADING EVENTS WITHOUT VALIDATING CURRENT STATE	INFORMATIONAL	SOLVED - 03/08/2026
REDUNDANT UPGRADETO() WRAPPER SHOULD BE REMOVED IN FAVOR OF DIRECTLY USING THE INHERITED UPGRADETOANDCALL()	INFORMATIONAL	SOLVED - 03/08/2026

7. FINDINGS & TECH DETAILS

7.1 ERC20 APPROVAL RACE CONDITION ALLOWS A SPENDER TO EXCEED THE INTENDED ALLOWANCE

// LOW

Description

The `DGLDToken.sol::approve()` function is intended to allow token holders to authorize a spender to transfer tokens on their behalf. However, the standard ERC20 `approve()` mechanism is vulnerable to a well-known front-running race condition where a spender can exploit an allowance update to spend more tokens than the owner intended.

This occurs because when an owner wants to change an existing allowance from amount `X` to amount `Y`, they must submit a new `approve()` transaction. A malicious spender monitoring the mempool can front-run this transaction by spending the original allowance `X` before the new transaction is confirmed, and then spending the new allowance `Y` after, resulting in a total spend of `X + Y` instead of the intended `Y`.

BVSS

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

Recommendation

Implement `increaseAllowance()` and `decreaseAllowance()` functions so that owners can adjust allowances relatively rather than absolutely, eliminating the race condition window.

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by implementing `increaseAllowance()` and `decreaseAllowance().`

Remediation Hash

https://github.com/goldtokensa/gtsa-base-smart-contracts/commit/fb9f4c68f0acfd5fbf64ea9f7b5c04e97553418c

7.2 SINGLE-STEP ADMIN TRANSFER IN CHANGEROLESADMIN() CAN PERMANENTLY LOCK ALL PRIVILEGED ROLES

// LOW

Description

The `DGLDToken.sol::changeRolesAdmin()` function is intended to allow the current `DEFAULT_ADMIN_ROLE` holder to transfer administrative control to a new address. However, the transfer is performed atomically in a single transaction, the new admin is granted the role and the current admin is revoked in the same call, with no confirmation step from the new admin's side.

If `newAdmin` is set to an incorrect address (e.g. a mistyped address, a contract that cannot interact with the token, or an EOA whose private key is lost), the `DEFAULT_ADMIN_ROLE` is permanently and irrecoverably transferred to an address that cannot exercise it. Since `DEFAULT_ADMIN_ROLE` is the admin for all other roles (`PAUSER_ROLE` and `BLACKLISTER_ROLE`) and is the only role authorized to approve contract upgrades via `_authorizeUpgrade()`, losing it means no new pausers or blacklisters can ever be granted or revoked.

BVSS

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

Recommendation

Implement a two-step transfer pattern where the current admin proposes a new admin, and the new admin must explicitly accept the role in a separate transaction.

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by implementing a two-step transfer pattern.

Remediation Hash

https://github.com/goldtokensa/gtsa-base-smart-contracts/commit/a6c2afea9434c89abcdd8ce864f20bab756825e0

7.3 IMMUTABLE BRIDGE ADDRESS IN __UPGRADEABLEOPTIMISMMINTABLEERC20_INIT() PREVENTS RECOVERY FROM BRIDGE MIGRATION OR MISCONFIGURATION

// LOW

Description

The `UpgradeableOptimismMintableERC20.sol::__UpgradeableOptimismMintableERC20_init()` function sets the `bridge` address once during initialization and stores it in namespaced storage with no setter function to update it post-deployment. The `bridge` address is the only address authorized to call `mint()` and `burn()` via the `onlyBridge` modifier, making it the most critical operational dependency of the token. However, there is no mechanism to update the bridge address short of performing a full contract upgrade via the UUPS proxy.

BVSS

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

Recommendation

Introduce a privileged setter function restricted to `DEFAULT_ADMIN_ROLE` that allows the bridge address to be updated without requiring a full contract upgrade.

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by implementing a setter function.

Remediation Hash

https://github.com/goldtokensa/gtsa-base-smart-contracts/commit/84a862a8e1f1c109e1d559fc1bdfe9e2d42c93d3

7.4 MISSING ZERO ADDRESS VALIDATION FOR _BRIDGE AND _REMOTETOKEN AND MISSING ZERO AMOUNT VALIDATION IN MINT() AND BURN()

// INFORMATIONAL

Description

Two related missing input validation issues exist across `DGLDToken.sol` and `UpgradeableOptimismMintableERC20.sol`:

- First, the `DGLDToken.sol::initialize()` function correctly validates that `_admin`, `_pauser`, and `_blacklister` are not zero addresses but does not apply the same validation to `_bridge` and `_remoteToken` before passing them to `__UpgradeableOptimismMintableERC20_init()`.
- Second, `UpgradeableOptimismMintableERC20.sol::mint()` and `burn()` do not validate that `_amount` is greater than zero before executing

While both are restricted to `onlyBridge` preventing arbitrary exploitation, a zero-amount call would silently succeed with no state change while still emitting `Mint` and `Burn` events with `amount = 0.`

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

Add zero address validation for `_bridge` and `_remoteToken` in `DGLDToken.sol::initialize()` and as a defense-in-depth measure inside `__UpgradeableOptimismMintableERC20_init()` as well.

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by adding validation.

Remediation Hash

https://github.com/goldtokensa/gtsa-base-smart-contracts/commit/24a16a570839e16696470e2ab8f821b4114d5f2d

7.5 REDUNDANT BLACKLIST AND UNBLACKLIST OPERATIONS EMIT MISLEADING EVENTS WITHOUT VALIDATING CURRENT STATE

// INFORMATIONAL

Description

The `Blacklistable.sol::blacklist()` and `Blacklistable.sol::unBlacklist()` functions do not validate the current blacklist state of `_account` before executing. `blacklist()` unconditionally sets `blacklisted[_account] = true` and emits a `Blacklisted` event even if the address is already blacklisted, and symmetrically `unBlacklist()` unconditionally sets `blacklisted[_account] = false` and emits an `UnBlacklisted` event even if the address was never blacklisted in the first place.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

Add a state check at the top of both functions to revert if the operation is a no-op.

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by following the given recommendation.

Remediation Hash

https://github.com/goldtokensa/gtsa-base-smart-contracts/commit/adbe37fa0003f69d4cebebd77d9eb4549df0a94a

7.6 REDUNDANT UPGRADETO() WRAPPER SHOULD BE REMOVED IN FAVOR OF DIRECTLY USING THE INHERITED UPGRADETOANDCALL()

// INFORMATIONAL

Description

The `DGLDToken.sol::upgradeTo()` function is a convenience wrapper around `upgradeToAndCall()` with empty calldata, which is already available and inherited from `UUPSUpgradeable`. The function provides no additional logic, safety checks, or functionality beyond what `upgradeToAndCall()` already offers, it simply calls it with an empty bytes argument. Having a redundant wrapper function in the codebase introduces unnecessary code surface.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

Remove the `upgradeTo()` wrapper entirely from `DGLDToken.sol` and rely solely on the inherited `upgradeToAndCall()` from `UUPSUpgradeable` for all upgrade operations.

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by removing `upgradeTo().`

Remediation Hash

https://github.com/goldtokensa/gtsa-base-smart-contracts/commit/7ee11d642083882334beddefc3f754c1ce0876fc

---

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.