// Security Assessment 02.27.2026 - 03.02.2026

V2 Working
Gold Token SA (MKS Pamp)

HALBORN

V2 Working - Gold Token SA (MKS Pamp)

Prepared by: HALBORN

Last Updated 03/16/2026

Date of Engagement: February 27th, 2026 - March 2nd, 2026

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	0	0	0	3	4

TABLE OF CONTENTS

1. INTRODUCTION

`Gold Token` engaged `Halborn` to perform a security assessment of their smart contracts starting on February 27th 2026 and ending on March 2nd, 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 `DGLDTokenNoFee` contract implements the core V3 gold-backed ERC20 token logic for the DGLD protocol, designed to tokenize physical gold bars on-chain by minting proportional ERC20 tokens against collateralized gold bar IDs via `mint()`, and redeeming them back through `burn()`. Each gold bar is registered by its unique `barId` mapped via `sha256` to its fine weight in 18-decimal token units, with `barCount` tracking total collateralized bars.

2. ASSESSMENT SUMMARY

`Halborn` was allocated 2 days for this engagement and assigned 1 full-time security engineer to conduct a comprehensive review of the smart contracts within scope. The engineer is an expert 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 mostly addressed by the `Gold Token` team. The main recommendations were:

- `Use ceiling division for both fee calculations.`
- `Restore the setFeeAddress() function in DGLDTokenNoFee.`
- `Restore the _transfer() override in DGLDTokenV3 to re-enforce blacklist and pause checks.`

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-contracts

(b) Assessed Commit ID: 5f6a7a6

(c) Items in scope:

- contracts/DGLDTokenNoFees.sol
- contracts/DGLDTokenV3.sol
- contracts/Blacklistable.sol

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

REMEDIATION COMMIT ID:

- 8a495a1
- 173559f
- 0d38ae7
- ae84cec
- 7f95189

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
4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
FEE AMOUNT TRUNCATION ALLOWS REDEMPTION AND CREATION FEE EVASION THROUGH SMALL AMOUNT OPERATIONS	LOW	SOLVED - 03/04/2026
NO MECHANISM TO UPDATE FEEADDRESS IN V3 LEAVES PROTOCOL WITH NO FEE RECOVERY PATH	LOW	SOLVED - 03/02/2026
REMOVAL OF _TRANSFER OVERRIDE IN DGLDTOKENNOFEE ALLOWS FEE TRANSFERS TO AND FROM BLACKLISTED ADDRESSES	LOW	SOLVED - 03/04/2026
GETBARWEIGHT() SILENTLY RETURNS ZERO FOR NON-EXISTENT BAR IDS	INFORMATIONAL	ACKNOWLEDGED - 03/06/2026
MISLEADING NATSPEC COMMENT	INFORMATIONAL	SOLVED - 03/02/2026
OUTDATED SOLIDITY COMPILER VERSION	INFORMATIONAL	ACKNOWLEDGED - 03/06/2026
INCORRECT FEE GUARD IN MINT() TRIGGERS ZERO-VALUE ERC20 TRANSFER AND PRODUCES MISLEADING ON-CHAIN EVENTS	INFORMATIONAL	SOLVED - 03/02/2026

7. FINDINGS & TECH DETAILS

7.1 FEE AMOUNT TRUNCATION ALLOWS REDEMPTION AND CREATION FEE EVASION THROUGH SMALL AMOUNT OPERATIONS

// LOW

Description

The `DGLDTokenNoFee.burn()` and `DGLDTokenNoFee.mint()` functions compute redemption and creation fees using integer division without rounding up. However, Solidity integer division always truncates towards zero, meaning for sufficiently small `barWeight` values the fee calculation rounds down to zero, allowing a redeemer or minter to avoid paying fees entirely.

Scenario:

1. Minter mints a bar with `amount = 499 wei` and `creationFee = 20.`
2. Fee calculation: `(499 * 20) / 10_000 = 0` , zero creation fee paid.
3. Burner later burns the same bar with `redemptionFee = 20`
4. Fee calculation: `(499 * 20) / 10_000 = 0` , zero redemption fee paid.
5. Both operations complete with no fees collected by `feeAddress.`

BVSS

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

Recommendation

Use ceiling division for both fee calculations.

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by rounding up the fee amount.

Remediation Hash

https://github.com/goldtokensa/gtsa-contracts/commit/8a495a14babeacdd79650a3da522ff2353a38a7b

7.2 NO MECHANISM TO UPDATE FEEADDRESS IN V3 LEAVES PROTOCOL WITH NO FEE RECOVERY PATH

// LOW

Description

The `DGLDTokenNoFee` contract retains `feeAddress` as a storage variable inherited from V1 but removes the `setFeeAddress()` function that previously allowed updating it via `SET_FEE_ADDRESS_ROLE`. However, `creationFee = 20` and `redemptionFee = 20` are actively used in V3 mint and burn operations, meaning fee transfers to `feeAddress` occur on every mint and burn call.

Due to this if the protocol wants to change fee address due to any reasons, they won't be able to do so as no function exists in V3 to update `feeAddress`.

BVSS

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

Recommendation

Restore the `setFeeAddress()` function in `DGLDTokenNoFee` .

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by adding a setter function to change `feeAddress`.

Remediation Hash

https://github.com/goldtokensa/gtsa-contracts/commit/173559ff87e20e4776fc420815fd65c15f245104

7.3 REMOVAL OF _TRANSFER OVERRIDE IN DGLDTOKENNOFEE ALLOWS FEE TRANSFERS TO AND FROM BLACKLISTED ADDRESSES

// LOW

Description

The `DGLDTokenV3` enforces blacklist checks on all public-facing ERC20 functions (`transfer()`, `transferFrom()`, `approve()`) and at the `_mint()` and `_burn()` internal level. However, the `_transfer()` internal override present in `DGLDTokenV2` was not carried forward into `DGLDTokenV3`, meaning any internal `_transfer()` call made directly within the contract logic bypasses blacklist and pause checks entirely.

The `creationFee = 20` (0.20%) and `redemptionFee = 20` (0.20%) are actively passed during `mint()` and `burn()` operations, triggering internal `_transfer()` calls to `feeAddress` on every mint and burn. So now even if feeAddress is blacklisted then too the tokens would get transferred without reverting.

BVSS

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

Recommendation

Restore the `_transfer()` override in `DGLDTokenV3` to re-enforce blacklist and pause checks on all internal token movements including fee transfers.

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by overriding `_transfer.`

Remediation Hash

https://github.com/goldtokensa/gtsa-contracts/commit/0d38ae74d92ce8b6fdf3acec58a672ad8f011390

7.4 GETBARWEIGHT() SILENTLY RETURNS ZERO FOR NON-EXISTENT BAR IDS

// INFORMATIONAL

Description

The `DGLDTokenNoFee.getBarWeight()` function is intended to return the fine weight of a registered gold bar. However, it does not revert when queried with a non-existent `barId`, instead silently returning `0`. This makes it impossible for callers to distinguish between a bar that does not exist and a bar that was registered with zero weight. Since Solidity mappings return the zero value for uninitialised keys, any non-existent `barId` will return `0` without any indication of failure.

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 an explicit existence check and revert with a meaningful error for non-existent bars.

Remediation Comment

ACKNOWLEDGED: The **Gold Token team** made a business decision to acknowledge this finding and not alter the contracts.

7.5 MISLEADING NATSPEC COMMENT

// INFORMATIONAL

Description

The `DGLDTokenNoFee.transferFrom()` function retains a NatSpec comment inherited from the V1 `DGLDToken` contract stating it applies custody fees. However, all custody fee logic was deliberately removed in `DGLDTokenNoFee` as part of the V3 upgrade. The function now simply delegates to `super.transferFrom()` with no fee logic whatsoever. The same issue exists on `transfer().`

BVSS

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

Recommendation

Update the NatSpec comments on both functions to accurately reflect V3 behaviour.

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by removing the misleading comment.

Remediation Hash

https://github.com/goldtokensa/gtsa-contracts/commit/ae84cecd8c6ffafd4a4b91d06101c57410c46d82

7.6 OUTDATED SOLIDITY COMPILER VERSION

// INFORMATIONAL

Description

All contracts in the DGLD protocol are compiled with Solidity `0.8.4`, which is a outdated compiler version. Newer compiler versions include important bug fixes, security patches, gas optimisations, and improved error reporting that are not available in `0.8.4`.

BVSS

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

Recommendation

Upgrade all contracts to the latest stable Solidity release.

Remediation Comment

ACKNOWLEDGED: The **Gold Token team** made a business decision to acknowledge this finding and not alter the contracts.

7.7 INCORRECT FEE GUARD IN MINT() TRIGGERS ZERO-VALUE ERC20 TRANSFER AND PRODUCES MISLEADING ON-CHAIN EVENTS

// INFORMATIONAL

Description

The `DGLDTokenNoFee.mint()` function contains an inconsistent guard condition when executing the creation fee transfer. The function computes `creationFeeAmount` via integer division which can truncate to zero for small `amount` values, but then checks `if (creationFee > 0),` the raw basis point rate, instead of `if (creationFeeAmount > 0)`, the computed transfer amount before calling `_transfer()`. This means whenever truncation reduces `creationFeeAmount` to zero, the guard still passes and `_transfer(to, feeAddress, 0)` is invoked unnecessarily.

BVSS

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

Recommendation

Fix the guard in `mint()` to check the computed amount, not the fee rate.

Remediation Comment

SOLVED: The **Gold Token team** solved this finding by changing `creationFee` to `creationFeeAmount` inside `mint().`

Remediation Hash

https://github.com/goldtokensa/gtsa-contracts/commit/7f9518981546f0884b2287e87e01822403c9224d

---

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.