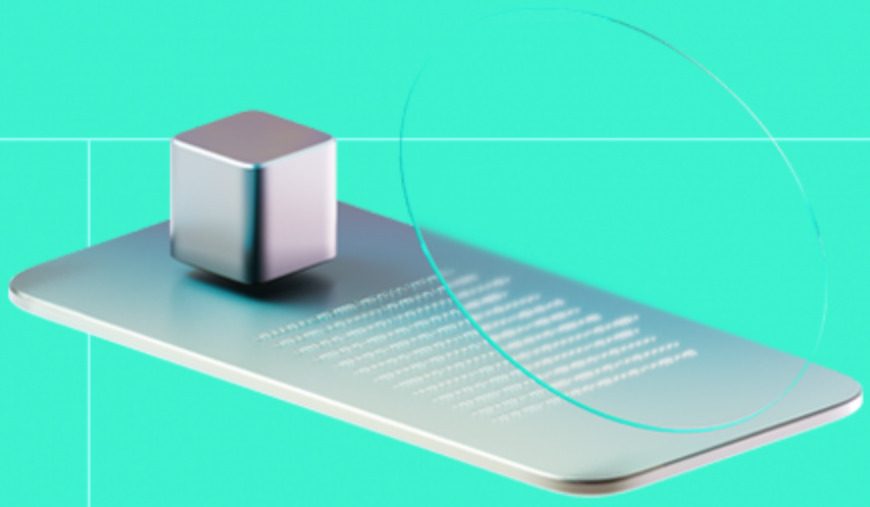
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Gold Token SA

Date: 12/03/2026

We express our gratitude to the Gold Token SA team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Gold Token is a permissioned, upgradeable ERC20 digital-gold system where token supply is managed through controlled issuance and redemption tied to bar identifiers.

Document

Name	Smart Contract Code Review and Security Analysis Report for Gold Token SA
Audited By	Ivan Bondar; Olesia Bilenka
Approved By	Khrystyna Tkachuk
Website	https://www.gtsa.io
Changelog	03/03/2026 - Preliminary Report
	12/03/2026 - Final Report
Platform	Ethereum
Language	Solidity
Tags	Fungible Token; Real World Assets (RWA); Centralization; Upgradable
Methodology	https://docs.hacken.io/methodologies/smart-contracts

Review Scope

Repository	https://github.com/goldtokensa/gtsa-contracts/tree/v2-working
Commit	5f6a7a6
Final commit	76f535a

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

6	6	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	2
Low	0

Vulnerability	Severity	Status
F-2026-15264 - Creation Fee Transferred to Deprecated feeAddress	Medium	Fixed
F-2026-15265 - Blacklisted User's Funds Are Permanently Frozen With No Recovery Mechanism	Medium	Fixed
F-2026-15267 - File Name/Contract Name Mismatch (DGLDTokenNoFees.sol vs. DGLDTokenNoFee)	Info	Fixed
F-2026-15268 - Redundant sha256 Computations in Burn Flow Increase Gas Usage	Info	Fixed
F-2026-15269 - Unused Imports in DGLDTokenNoFees.sol Increase Noise and Maintenance Risk	Info	Fixed
F-2026-15270 - Deprecated Role Constants Are Retained Without Active Role-Controlled Functions	Info	Fixed

Documentation quality

- Functional requirements are detailed.
  - Project overview is detailed
  - All roles in the system are described.
  - Use cases are described and detailed.
  - For each contract, all futures are described.
  - All interactions are described.
- Technical description is detailed.
  - Run instructions are provided.
  - Technical specification is provided.
  - The NatSpec documentation is sufficient.

Code quality

- The development environment is configured.

Test coverage

Code coverage of the project is **89%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is present.
- Interactions by several users are tested.

Table of Contents

System Overview **6**

Privileged Roles 6

Potential Risks **7**

Findings **8**

Vulnerability Details 8

Disclaimers 22

Appendix 1. Definitions **23**

Severities 23

Potential Risks 23

Appendix 2. Scope **24**

Appendix 3. Additional Valuables **25**

System Overview

Gold Token is a permissioned, upgradeable ERC20 digital-gold system where token supply is managed through controlled issuance and redemption tied to bar identifiers.

- **DGLDTokenV3.sol:** main token surface that enforces operational controls (pause + blacklist checks) on transfers, approvals, allowance changes, minting, and burning.
- **DGLDTokenNoFees.sol:** core issuance/redemption engine; records collateralized bars via barId, maintains barCount, executes mint/burn flows, and stores the T&C URL.
- **Blacklistable.sol:** compliance extension providing address-level blacklist management and transfer-gating via notBlacklisted.

Operationally, designated roles mint tokens when bars are onboarded and burn tokens when bars are redeemed, while day-to-day token activity remains subject to compliance and emergency controls.

Privileged roles

- `DEFAULT_ADMIN_ROLE`: top-level authority for role management (grant/revoke/admin of privileged roles).
- `PAUSER_ROLE`: can pause and unpause protocol token operations.
- `MINTER_ROLE`: can mint tokens for new bar records.
- `BURNER_ROLE`: can burn tokens during redemption flows.
- `SET_TC_ROLE`: can update the Terms & Conditions URL.
- `BLACKLISTER_ROLE`: can blacklist and unblacklist addresses.

Potential Risks

- **Scope Definition and Security Guarantees:** The audit does not cover all code in the repository. Contracts outside the audit scope may introduce vulnerabilities, potentially impacting the overall security due to the interconnected nature of smart contracts.
- **Upgradability and Future Version Modifications:** The contract's upgradable nature means that its implementation can be modified in future versions. This flexibility allows for necessary updates and improvements but also introduces uncertainty for users regarding future changes in the contract's behavior and rules. Users should stay informed about any contract upgrades and understand their implications.
- **Role/Governance Concentration:** `DEFAULT_ADMIN_ROLE` can manage other privileged roles. Compromise, coercion, or operational error at this layer can lead to full control over minting/burning operators, pause authority, and blacklist authority.
- **Mint/Burn Operational Trust and Collateralization:** `MINTER_ROLE` and `BURNER_ROLE` control issuance/redemption execution. Users must trust off-chain processes to correctly validate bar custody, bar identity, and redemption workflow; on-chain logic cannot independently prove physical collateral quality or existence.
- **Pause-Induced Availability:** `PAUSER_ROLE` can stop core token actions. This is useful for incident response but can also create temporary loss of transferability and reduced protocol liveness during pauses.
- **Blacklist Censorship/Freezing:** `BLACKLISTER_ROLE` can block addresses from interacting with token functions guarded by blacklist checks. This supports compliance objectives but introduces explicit censorship/funds-mobility restrictions for affected users.
- **Terms & Conditions Reference:** `SET_TC_ROLE` can update `tcURL`. Legal/policy references may change over time, so users and integrators must monitor updates to avoid relying on stale assumptions.

Findings

Vulnerability Details

F-2026-15264 - Creation Fee Transferred to Deprecated feeAddress - Medium

Description:

Fee transfer logic remains active in V3 through **DGLDTokenNoFee**, while the fee destination relies on a deprecated storage variable that is no longer operationally managed in the upgrade path. Non-zero creation or redemption fees can therefore be routed to an inaccessible legacy address, or trigger `mint`/`burn` failure when the destination is unset.

Legacy fee-related storage is retained for layout compatibility, including `feeAddress`, but fee transfers are still executed in core token flows:

DGLDTokenNoFees:

```
uint256 public custodyFee; /// @notice Deprecated
uint256 public transferFee; /// @notice Deprecated

// Tracks the number of gold bar minted
uint256 public barCount;

address public feeAddress; /// @notice Deprecated
bool public feeActivated; /// @notice Deprecated
```

In `mint`, a non-zero creationFee causes transfer of `creationFeeAmount` to `feeAddress`:

```
uint256 creationFeeAmount = amount > 0
    ? (amount * creationFee) / 10000
    : 0;

_mint(to, amount);
/* ... */
if (creationFee > 0) {
    _transfer(to, feeAddress, creationFeeAmount);
}
```

In `burn`, a non-zero `redemptionFeeAmount` is transferred to the same destination:

```
uint256 redemptionFeeAmount = burnAmount > 0
    ? (burnAmount * redemptionFee) / 10000
    : 0;

_burn(from, burnAmount);
emit Burn(from, burnAmount, barId, redemptionFeeAmount);
if (redemptionFeeAmount > 0) _transfer(from, feeAddress, redemptionFeeAmount)
;
```

V3 inherits from **DGLDTokenNoFee** and does not add active fee-address management in the shown inheritance path:

```
contract DGLDTokenV3 is DGLDTokenNoFee, PausableUpgradeable, Blacklistable {.
..}
```

Under these conditions, non-zero fee usage introduces two concrete outcomes in the same execution path: fee assets may be irrecoverably transferred to a stale or inaccessible legacy address, and `mint`/`burn` operations may revert when `feeAddress` is zero due to **ERC20** transfer constraints. The issue affects every `mint`/`burn` operation where non-zero fee inputs are provided.

Assets:

- contracts/DGLDTokenNoFees.sol [https://github.com/goldtokensa/gtsa-contracts]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 3/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Fee behavior should be aligned with the no-fee upgrade objective by fully disabling fee transfers in `mint` and `burn`, while keeping deprecated storage variables only for storage-layout preservation. Fee inputs can be strictly rejected (`== 0`) or ignored with explicit event semantics, so dependency on

deprecated `feeAddress` state is eliminated and both irreversible fee transfers and fee-path execution failures are removed.

Resolution:

Fixed in `173559f`:

The **DGLDTokenMarch2026** base contract (formerly **DGLDTokenNoFee**, renamed in `76f535a`) now includes a role-restricted `setFeeAddress` function gated by `SET_FEE_ADDRESS_ROLE`, with an explicit zero-address guard. The `feeAddress` storage variable is no longer annotated as deprecated and is treated as an actively managed operational field. Unused legacy role constants (`ADD_WHITELIST_ROLE`, `REMOVE_WHITELIST_ROLE`, `SET_CUSTODY_FEE_ROLE`, `SET_TRANSFER_FEE_ROLE`) were removed in `787cf94`, while `SET_FEE_ADDRESS_ROLE` was retained to support ongoing fee address governance.

```
address public feeAddress;

function setFeeAddress(address value)
    public
    onlyRole(SET_FEE_ADDRESS_ROLE)
{
    require(
        value != address(0),
        "DGLD: fee address connot be address zero"
    );
    address oldFeeAddress = feeAddress;
    feeAddress = value;
    emit FeeAddressUpdated(oldFeeAddress, feeAddress);
}
```

The mint fee guard was corrected to evaluate `creationFeeAmount > 0` instead of `creationFee > 0` in `7f95189`, and the contract naming was updated to remove the misleading "no-fee" label.

```
if (creationFeeAmount > 0) {
    _transfer(to, feeAddress, creationFeeAmount);
}
```

F-2026-15265 - Blacklisted User's Funds Are Permanently Frozen With No Recovery Mechanism - Medium

Description:

Blacklist enforcement prevents blacklisted accounts from transferring or redeeming tokens, yet no dedicated administrative recovery flow exists for unresolved frozen balances. When an address cannot be safely unblacklisted, corresponding gold-backed token balances can remain permanently immobilized.

DGLDTokenV3 enforces `notBlacklisted` restrictions across transfer and burn execution paths, so a blacklisted holder cannot move funds through normal ERC20 transfer flow or redemption burn flow. The blacklist module exposes only state toggling via `blacklist` / `unblacklist` operations and does not provide a seizure, forced-transfer, or burn-from-blacklisted mechanism.

```
function blacklist(address _account) external virtual onlyRole(BLACKLISTER_ROLE) {
    blacklisted[_account] = true;
    emit Blacklisted(_account);
}

function unBlacklist(address _account) external virtual onlyRole(BLACKLISTER_ROLE) {
    blacklisted[_account] = false;
    emit UnBlacklisted(_account);
}
```

Given this design, asset movement from a blacklisted address is operationally dependent on unblacklisting. If unblacklisting is not permissible due to compliance, legal, or policy constraints, frozen balances cannot be resolved on-chain, creating a persistent mismatch between usable circulating token supply and off-chain backing operations tied to redemption.

Assets:

- contracts/DGLDTokenV3.sol [https://github.com/goldtokensa/gtsa-contracts]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: A controlled blacklisted-funds resolution mechanism should be introduced for exceptional cases, such as administrative burn, forced transfer to a governed recovery address, or regulated escrow migration. The mechanism should be role-restricted, event-complete, and explicitly documented to support operation during paused conditions when emergency handling is required. Governance and compliance procedures should define when each recovery path is permitted to preserve blacklist intent while preventing permanent asset deadlock.

Resolution: Fixed in `17b7c27`:

The **DGLDTokenV3** contract now implements a dedicated token recovery mechanism for blacklisted addresses, directly addressing the absence of an administrative resolution flow for frozen balances.

A role-restricted `recoverTokensFromBlacklistedAddress` function, gated by the `TOKEN_RECOVERER_ROLE`, transfers the full token balance of a blacklisted address to a configurable `recoveryAddress`.

```
function recoverTokensFromBlacklistedAddress(address _blackListedAddress)
        external
        whenNotPaused
        onlyRole(TOKEN_RECOVERER_ROLE)
    {
        // Ensure address is blacklisted
        if (!blacklisted[_blackListedAddress]) {
            revert NotBlacklisted();
        }


        // Transfer tokens to recovery address
        uint256 amount = balanceOf(_blackListedAddress);
        _transfer(_blackListedAddress, recoveryAddress, amount);


        emit RecoveryFromBlacklistedAddress(_blackListedAddress, recoveryAddr
ess, amount);
    }
```

The `_transfer` internal override has been modified to permit transfers originating from a blacklisted address when the destination is the designated recoveryAddress, while continuing to revert for all other transfer targets.

```
function _transfer(
    address from,
    address to,
    uint256 amount
)
    internal
    virtual
    override
    whenNotPaused
    notBlacklisted(_msgSender())
    notBlacklisted(to)
{
    if (blacklisted[from] && to != recoveryAddress) {
        revert AccountBlacklisted(from);
    }
    super._transfer(from, to, amount);
}
```

The recovery address is configurable via `setRecoveryAddress`, restricted to `DEFAULT_ADMIN_ROLE`, and protected against assignment to the zero address.

```
function setRecoveryAddress(address value) public onlyRole(DEFAULT_ADMIN_ROLE
) {
    if (value == address(0)) {
        revert RecoveryAddressZero();
    }

    address oldRecoveryAddress = recoveryAddress;
    recoveryAddress = value;
    emit RecoveryAddressUpdated(oldRecoveryAddress, recoveryAddress);
}
```

F-2026-15267 - File Name/Contract Name Mismatch (DGLDTokenNoFees.sol vs. DGLDTokenNoFee) - Info

Description: The file is named **DGLDTokenNoFees.sol** (plural), while the contract inside is `DGLDTokenNoFee` (singular).

This inconsistency is not an on-chain vulnerability by itself, but it is a maintenance and integration risk, which:

- Increases probability of integration errors in scripts/tooling (especially codegen and artifact mapping).
- Raises risk of importing the wrong artifact in multi-version repositories.

Assets:

- contracts/DGLDTokenNoFees.sol [https://github.com/goldtokensa/gtsa-contracts]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Align naming convention in one direction and keep it consistent across:

- Solidity file name
- Contract name
- Deployment scripts
- Documentation references

Resolution: Fixed in `07a0e62`:

The **DGLDTokenNoFees.sol** was renamed to **DGLDTokenNoFee.sol**, eliminating the singular/plural mismatch between file name and contract name. V3 inheritance references were updated in the same change, removing artifact and import ambiguity in the active upgrade path.

Subsequent commit `76f535a` performed a coordinated rename to **DGLDTokenMarch2026** for both file and contract, preserving naming consistency across source file, contract declaration, and import usage.

F-2026-15268 - Redundant sha256 Computations in Burn Flow Increase Gas Usage - Info

Description:

The same hash value `sha256(abi.encodePacked(barId))` is recomputed multiple times within a single burn path, instead of being cached once and reused.

```
require(barIds[sha256(abi.encodePacked(barId))] > 0, "Bar id not found");
require(
    redemptionFee <= 10_000,
    "canBurn: redemption fee percentage too high"
);

uint256 barWeight = barIds[sha256(abi.encodePacked(barId))];

...

uint256 burnAmount = barIds[sha256(abi.encodePacked(barId))];
delete barIds[sha256(abi.encodePacked(barId))];
```

For one `burn` call:

- `_canBurn` computes the hash multiple times
- `burn` computes it again for read + delete

This leads to unnecessary gas overhead on a frequently used operational path.

Assets:

- contracts/DGLDTokenNoFees.sol [https://github.com/goldtokensa/gtsa-contracts]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Cache the bar hash once per function and reuse it.

Resolution: Fixed in `24338d6`:

The `burn` function now computes `sha256(abi.encodePacked(barId))` once into a local `barIdHash` variable and reuses it:

```
bytes32 barIdHash = sha256(abi.encodePacked(barId));
require(_canBurn(from, barIdHash, redemptionFee),"Insufficient funds");

uint256 burnAmount = barIds[barIdHash];
delete barIds[barIdHash];
```

F-2026-15269 - Unused Imports in DGLDTokenNoFees.sol Increase Noise and Maintenance Risk - Info

Description:

The **DGLDTokenNoFee.sol** file imports **Initializable** and **MathUpgradeable**, but neither is directly used in **DGLDTokenNoFee**:

```
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
import "@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol";
```

- **MathUpgradeable** has no usages in the contract.
- **Initializable** is not explicitly referenced either.

This increases code noise and may mislead reviewers/integrators into thinking initializer/math helpers are used here.

Assets:

- contracts/DGLDTokenNoFees.sol [https://github.com/goldtokensa/gtsa-contracts]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

Remove unused imports:

- Remove **Initializable** import if not needed explicitly in this file.
- Remove **MathUpgradeable** import unless math helpers are reintroduced.

Resolution:

Fixed in `fb37b02`:

The unused **Initializable** import has been removed from the base contract (now **DGLDTokenMarch2026.sol**). The **MathUpgradeable** import, which

was also flagged as unused at the time the issue was submitted, is now actively referenced in the contract.

F-2026-15270 - Deprecated Role Constants Are Retained Without Active Role-Controlled Functions - Info

Description:

Several legacy role constants remain defined in **DGLDTokenNoFee** contract, but there are no corresponding public functions in V3 path that use these roles.

```
bytes32 public constant ADD_WHITELIST_ROLE = keccak256("ADD_WHITELIST_ROLE");
bytes32 public constant REMOVE_WHITELIST_ROLE = keccak256("REMOVE_WHITELIST_R
OLE");
bytes32 public constant SET_CUSTODY_FEE_ROLE = keccak256("SET_CUSTODY_FEE_ROL
E");
bytes32 public constant SET_FEE_ADDRESS_ROLE = keccak256("SET_FEE_ADDRESS_ROL
E");
bytes32 public constant SET_TRANSFER_FEE_ROLE = keccak256("SET_TRANSFER_FEE_R
OLE");
```

This can confuse operators/integrators (roles appear to exist but provide no effective capability).

These are public constant values, so they do not consume storage slots. Removing them would not break storage during upgrades,

Assets:

- contracts/DGLDTokenNoFees.sol [https://github.com/goldtokensa/gtsa-contracts]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

- Consider removing unused constants.

- If they matter for ABI compatibility, keep them and explicitly document as legacy/deprecated.

Resolution: Fixed in `787cf94`:

The four unused legacy role constants — `ADD_WHITELIST_ROLE`, `REMOVE_WHITELIST_ROLE`, `SET_CUSTODY_FEE_ROLE`, and `SET_TRANSFER_FEE_ROLE` — have been removed from the **DGLDTokenMarch2026** base contract (formerly **DGLDTokenNoFee**). The remaining role constants (`MINTER_ROLE`, `BURNER_ROLE`, `SET_FEE_ADDRESS_ROLE`, `SET_TC_ROLE`) each have a corresponding role-gated function in the active V3 execution path. `SET_FEE_ADDRESS_ROLE` was retained because `setFeeAddress` is now an actively used administrative function. The removal eliminates operator and integrator confusion from role definitions that provided no effective capability.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

As part of Hacken's ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.
The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at https://hacken.io/audits/.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/goldtokensa/gtsa-contracts/tree/v2-working
Commit	5f6a7a6569ecf4080d9bc1cbf216ba9036f7ff27
Final Commit	76f535af8e3dd3dc8f1316e105eee11c1539c05a
Whitepaper	N/A
Requirements	NatSpec
Technical Requirements	README.md

Asset	Type
contracts/Blacklistable.sol [https://github.com/goldtokensa/gtsa-contracts]	Smart Contract
contracts/DGLDTokenNoFees.sol [https://github.com/goldtokensa/gtsa-contracts]	Smart Contract
contracts/DGLDTokenV3.sol [https://github.com/goldtokensa/gtsa-contracts]	Smart Contract

Appendix 3. Additional Valuables

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.

Frameworks and Methodologies

This security assessment was conducted in alignment with recognised penetration testing standards, methodologies and guidelines, including the NIST SP 800-115 – Technical Guide to Information Security Testing and Assessment, and the Penetration Testing Execution Standard (PTES), These assets provide a structured foundation for planning, executing, and documenting technical evaluations such as vulnerability assessments, exploitation activities, and security code reviews. Hacken's internal penetration testing methodology extends these principles to Web2 and Web3 environments to ensure consistency, repeatability, and verifiable outcomes.