CERTIK

StakeMyGold

Security Assessment

CertiK Assessed on May 6th, 2026

CertiK Assessed on May 6th, 2026

StakeMyGold

The security assessment was prepared by CertiK.

Executive Summary

TYPES	ECOSYSTEM	METHODS
Staking	EVM Compatible	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE
Solidity	Preliminary comments published on 03/26/2026 Final report published on 05/06/2026

Vulnerability Summary

20	13	3	0	4	0
Total Findings	Resolved	Multi-Sig	Partially Resolved	Acknowledged	Declined

3	Centralization	3 Multi-Sig	Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.
0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
0	Major		Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
5	Medium	3 Resolved, 2 Acknowledged	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
9	Minor	7 Resolved, 2 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
3	Informational	3 Resolved	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | STAKEMYGOLD

CODEBASE | STAKEMYGOLD

Repository

https://github.com/ardata-tech/stakemygold-smart-contracts

https://github.com/ardata-tech/goldfish-stggbr

Commit

eced94d514e3ec847660b32528b4a2977abe9d3f

954455b1c3b129fa40036b8fb5441d952d730b30

52246af2b5a158c5116b7e47d21cd8d33b5ca271

fd2cbf743a4f7d036647b18083563caa6f632504

4b0a2b0bca93fc102347580305e47668947b7246

AUDIT SCOPE | STAKEMYGOLD

ardata-tech/stakemygold-smart-contracts

contracts/StakeMyGold.sol

ardata-tech/goldfish-stggbr

contracts/src/StakedGGBR.sol

contracts/src/YieldController.sol

contracts/src/StakedGGBR.sol

contracts/src/YieldController.sol

contracts/src/StakedGGBR.sol

contracts/src/YieldController.sol

APPROACH & METHODS | STAKEMYGOLD

This audit was conducted for StakeMyGold to evaluate the security and correctness of the smart contracts associated with the StakeMyGold project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Static Analysis, Formal Verification, and Manual Review.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

REVIEW NOTES | STAKEMYGOLD

Overview

StakeMyGold

UUPS-upgradeable ERC-20 staking for the GGBR token, administered with OpenZeppelin AccessControl. Users may hold several open stakes at once; each stake is a StakeEntry with its own id, principal, start time, lock duration, and APR snapshot taken at deposit. Lock lengths come from a period registry (defaults such as 90 / 183 / 365 days; admin can add periods).

User flow: stake pulls tokens into the contract; unstake accrues time-based rewards, may apply an early-withdrawal penalty on rewards, then sends principal and net rewards to the user. Off-chain settlement uses PrincipalWithdrawnToIONWallet to move principal to an ION wallet and depositFromION to return funds and update paid-to-ION accounting. Principal, rewards, penalties, and rehypothecation profit are tracked separately; penalty and profit balances can be swept to a dedicated penalty wallet.

StakedGGBR

ERC-4626 vault over GGBR with a share token (stGGBR) and AccessControl. The contract is deployed once and is not upgradeable via proxy. Share and asset conversion uses a virtual offset (decimals offset 6) to reduce first-depositor inflation risk. Standard ERC-4626 withdraw and redeem entrypoints are disabled (max withdraw and max redeem read as zero). Exits go through requestWithdraw or requestRedeem, which create a withdrawal request: shares are moved to the vault, remain locked for withdrawalDelay (default seven days, capped at thirty), then claimWithdrawal pays assets and burns shares, or cancelWithdrawal returns shares.

Yield enters only through depositYield, callable by the configured yieldController address. That path pulls GGBR from the controller, optionally sends a performance fee to the fee recipient, and increases vault assets and the exchange rate under normal ERC-4626 accounting.

YieldController

Immutable helper: the constructor fixes the vault and the GGBR token. Addresses with the yield operator role call injectYield, which pulls GGBR from the caller, approves the vault, and forwards into depositYield. The controller records injection history, enforces a minimum time between injections, and limits each injection relative to vault total assets (fifty percent maximum).

External dependencies

- OpenZeppelin Upgradeable modules:
  - `AccessControlUpgradeable`
  - `UUPSUpgradeable`
  - `Initializable`
  - `PausableUpgradeable`

  - `ReentrancyGuardUpgradeable`

- OpenZeppelin modules (non-upgradeable):
  - `ERC4626`
  - `ERC20`
  - `AccessControl`
  - `Pausable`
  - `Math`

- OpenZeppelin token utilities:
  - `IERC20`
  - `SafeERC20`

- Standards/assumptions:
  - ERC20-compatible behavior of `ggbrToken` / GGBR for `transfer` / `transferFrom` in staking and vault settlement flows.
  - Where ERC-4626 previews and `totalAssets()` apply, GGBR behaves as a standard ERC-20 (no rebasing, no fee-on-transfer).
  - `IStakedGGBR` as the vault interface used by YieldController (`asset`, `totalAssets`, `exchangeRate`, `depositYield`).
  - Yield operators must have granted sufficient allowance to YieldController for `injectYield` to pull GGBR.

- Trusted external components:
  - `ionWallet` as operational settlement wallet for outbound/inbound principal transfers.
  - `penaltyWallet` as recipient of aggregated penalty and rehypothecation-profit withdrawals.
  - Proxy upgrade authority controlled through the account(s) holding `DEFAULT_ADMIN_ROLE` (StakeMyGold UUPS).
  - `feeRecipient` and `yieldController` addresses configured on the vault; vault `ADMIN_ROLE` / `DEFAULT_ADMIN_ROLE` for parameters and roles.
  - `YIELD_OPERATOR_ROLE` and `ADMIN_ROLE` on YieldController for injections, pause, interval, and emergency withdrawal.

Privileged functions

In the **StakeMyGold** protocol, some privileged roles are adopted to ensure the dynamic runtime updates of the project, which are specified in **Centralization Related Risks**, **Centralized Control of Contract Upgrade** and **Centralized Withdrawal Risk** findings.

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to best serve the community.

It is also worth noting the potential drawbacks of these functions, which should be clearly stated through the client's action/plan. Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should also be considered to move to the execution queue of the `Timelock` contract.

Clarification: Missing Solvency Accounting

The protocol does not maintain a verifiable ledger identity tying together onchain assets, offchain ION exposure, and user-facing liabilities. In particular, there is no enforced reconciliation of the form: `onchain token balance + assets deployed to ION = user principal liabilities + accrued/promised rewards + realized penalties/profits`. As a result, the system cannot prove whether reported profit is actually realized, whether reserves are sufficient to satisfy withdrawals, or whether value has been silently shifted between user liabilities and operator-controlled balances.

FINDINGS | STAKEMYGOLD

20	0	3	0	5	9	3
Total Findings	Critical	Centralization	Major	Medium	Minor	Informational

This report has been prepared for StakeMyGold to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 20 issues were identified. Leveraging a combination of Static Analysis, Formal Verification & Manual Review the following findings were uncovered:

ID	Title	Category	Severity	Status
STA-09	**Centralization Risks**	**Centralization**	**Centralization**	**2/3 Multi-Sig**
STA-10	**Centralized Control Of Contract Upgrade**	**Centralization**	**Centralization**	**2/3 Multi-Sig**
STA-12	**Centralized Withdrawal Risk**	**Centralization**	**Centralization**	**2/3 Multi-Sig**
STA-01	Rewards Continuing Beyond Lock Period Leads To Unbounded Reward Drain	Design Issue	Medium	Resolved
STA-02	Rehypothecation Rate Reduction Leads To Permanent Unstake Reverts	Design Issue	Medium	Resolved
STA-03	User Principal Can Be Used To Fund Other Users' Rewards	Design Issue	Medium	Acknowledged
STA-11	Short-Term Pool Design Leads To Meaningless Product Segmentation	Design Issue, Financial Manipulation	Medium	Acknowledged
STA-13	Withdrawal Request Period Withdrawn Amount And Shares Are Still Included In Calculations	Design Issue	Medium	Resolved
STA-04	`stake()` Is Incompatible With Fee-On-Transfer Tokens	Coding Issue	Minor	Resolved
STA-05	`depositFromION()` Does Not Restrict Repayments To `ionWallet`	Logical Issue	Minor	Resolved

ID	Title	Category	Severity	Status
STA-06	Unbounded Per-User Stake Growth Can Cause Withdrawal DoS	Denial of Service	Minor	Resolved
STA-07	`getPlatformStats()` Can Revert Due To Underflow In Reserve Calculation	Logical Issue	Minor	Resolved
STA-14	Reward Dilution Caused By Non-Linear Time-Based Distribution Enabling Front-Running Of `depositYield()`	Design Issue	Minor	Acknowledged
STA-15	Missing Validity Checks In `claimWithdrawal()`	Logical Issue	Minor	Resolved
STA-16	Direct Fund Injection Via Transfer	Logical Issue	Minor	Acknowledged
STA-20	Issue Of Contract Yield Being Absorbed By Virtual Shares	Logical Issue	Minor	Resolved
STA-21	Dilution Of Existing Rewards When Funds Re-Enter The Pool Via `cancelWithdrawal()`	Logical Issue	Minor	Resolved
STA-08	Unused Variables	Coding Issue	Informational	Resolved
STA-17	Unit Inconsistency In `exchangeRate()` Calculation	Inconsistency	Informational	Resolved
STA-18	`StakedGGBR` Interface ID Not Included In `supportsInterface`	Design Issue	Informational	Resolved

STA-09 | Centralization Risks

Category	Severity	Location	Status
Centralization	**Centralization**		**2/3 Multi-Sig**

Description

StakedGGBR.sol

In the contract `StakedGGBR`, the role `ADMIN_ROLE` has authority over the following functions:

- `setMinimumDeposit()`
- `setPerformanceFee()`
- `setFeeRecipient()`
- `setYieldController()`
- `setWithdrawalDelay()`
- `pauseDeposits()`
- `unpauseDeposits()`
- `emergencyWithdraw()`

Any compromise to the `ADMIN_ROLE` account may allow an attacker to arbitrarily change core vault parameters, rotate trusted controller/fee endpoints, freeze or resume deposits at will, and extract tokens from the contract via emergency withdrawal.

In the contract `StakedGGBR`, the role `yieldController` has authority over the following functions:

- `depositYield()`

Any compromise to the `yieldController` account may allow an attacker to inject manipulated yield flows into the vault (including fee side-effects) and influence vault accounting/exchange-rate behavior.

In the contract `StakedGGBR`, the role `DEFAULT_ADMIN_ROLE` (inherited from `AccessControl`) has authority over the following functions:

- `grantRole()`
- `revokeRole()`

Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow an attacker to grant/revoke privileged roles (including `ADMIN_ROLE`), thereby taking full control of administrative operations.

YieldController.sol

In the contract `YieldController`, the role `ADMIN_ROLE` has authority over the following functions:

- `pause()`
- `unpause()`
- `setMinInjectionInterval()`
- `emergencyWithdraw()`

Any compromise to the `ADMIN_ROLE` account may allow an attacker to halt/resume yield injections, alter injection cadence constraints, and withdraw tokens from the controller (including GGBR when paused).

In the contract `YieldController`, the role `YIELD_OPERATOR_ROLE` has authority over the following functions:

- `injectYield()`

Any compromise to the `YIELD_OPERATOR_ROLE` account may allow an attacker to perform arbitrary yield injections (within configured limits), impacting vault exchange rate progression and protocol economics.

In the contract `YieldController`, the role `DEFAULT_ADMIN_ROLE` (inherited from `AccessControl`) has authority over the following functions:

- `grantRole()`
- `revokeRole()`

Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow an attacker to grant/revoke `ADMIN_ROLE` and `YIELD_OPERATOR_ROLE`, effectively seizing or disrupting full controller governance.

StakeMyGold.sol

In the contract `StakeMyGold`, the role `DEFAULT_ADMIN_ROLE` has authority over the following functions:

- `setIONWallet()`
- `setPenaltyWallet()`
- `setAPRForPeriod()`
- `addNewPeriod()`
- `setPenalty()`
- `setRehypothecationRate()`
- `setMinStakeAmount()`
- `setMaxStakeAmount()`
- `withdrawAllPenaltiedToken()`
- `withdrawAllRehypothecationProfit()`
- `pause()`
- `unpause()`
- `emergencyWithdraw()`

Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow an attacker to reconfigure staking parameters and economic rates, redirect settlement/penalty wallets, pause or resume user operations, and transfer out penalty/profit reserves or other contract-held tokens.

In the contract `StakeMyGold`, the role `OPERATOR_ROLE` has authority over the following functions:

- `PrincipalWithdrawnToIONWallet()`

Any compromise to the `OPERATOR_ROLE` account may allow an attacker to move principal liquidity from the contract to the configured ION wallet, creating settlement disruption and potential fund misdirection if the destination wallet is also compromised or maliciously set.

In the contract `StakeMyGold`, the role `DEFAULT_ADMIN_ROLE` (inherited from `AccessControlUpgradeable`) has authority over the following functions:

- `grantRole()`
- `revokeRole()`

Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow an attacker to grant/revoke sensitive roles (including `DEFAULT_ADMIN_ROLE` and `OPERATOR_ROLE`) and seize full administrative control.

In the contract `StakeMyGold`, the role `DEFAULT_ADMIN_ROLE` (inherited from `UUPSUpgradeable`, authorized via `_authorizeUpgrade`) has authority over the following functions:

- `upgradeToAndCall()`

Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow an attacker to upgrade the implementation to malicious logic and fully take over protocol behavior and fund flows.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND

- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[StakeMyGold, 05/06/2026]: The team acknowledged the issue and adopted the multi-sig solution at the current stage. The StakeMyGold has transferred the `DEFAULT_ADMIN_ROLE` to a Gnosis in transaction, the `OPERATOR_ROLE` to the same Gnosis in transaction

- Gnosis safe contract address: https://etherscan.io/address/0xf75FEa6Baa04bFdc2521fE0d428db0d128C26890
- The 2/3 multisign addresses:
  - 0x27cb9db3c65ae5e25c7ea8216008f1ff70ef43fa
  - 0x390e4447014818f43323efdb9b3fb47a24aba92a
  - 0x664a03060920ae0e02d6b4d7c421b7910e2d01ee
  -

The StakedGGBR has transferred the `ADMIN_ROLE` to a Gnosis in transaction, the `DEFAULT_ADMIN_ROLE` to the same Gnosis in transaction.

The YieldController has transferred the `ADMIN_ROLE` to the same Gnosis in transaction, the `DEFAULT_ADMIN_ROLE` to the same Gnosis in transaction, the `YIELD_OPERATOR_ROLE` to the same Gnosis in transaction

- Gnosis safe contract address: https://etherscan.io/address/0x51820e72ba73fdab0e4471d1a2b20cb9bf6ee658
- The 2/3 multisign addresses:
  - 0x27cb9db3c65ae5e25c7ea8216008f1ff70ef43fa
  - 0x390e4447014818f43323efdb9b3fb47a24aba92a

- 0x664a03060920ae0e02d6b4d7c421b7910e2d01ee

STA-10 | Centralized Control Of Contract Upgrade

Category	Severity	Location	Status
Centralization	**Centralization**	**contracts/StakeMyGold.sol (init): 17**	**2/3 Multi-Sig**

Description

In the contract `StakeMyGold` , the role `DEFAULT_ADMIN_ROLE` has the authority to update the implementation contract behind the proxy contract:

```
function _authorizeUpgrade(address newImplementation) internal override
onlyRole(DEFAULT_ADMIN_ROLE) {}
```

Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow a hacker to take advantage of this authority and change the implementation contract which is pointed by proxy and therefore execute potential malicious functionality in the implementation contract.

Recommendation

We recommend that the team make efforts to restrict access to the admin of the proxy contract. A strategy of combining a time-lock and a multi-signature (⅔, ⅗) wallet can be used to prevent a single point of failure due to a private key compromise. In addition, the team should be transparent and notify the community in advance whenever they plan to migrate to a new implementation contract.

Here are some feasible short-term and long-term suggestions that would mitigate the potential risk to a different level and suggestions that would permanently fully resolve the risk.

Short Term:

A combination of a time-lock and a multi signature (⅔, ⅗) wallet mitigate the risk by delaying the sensitive operation and avoiding a single point of key management failure.

- A time-lock with reasonable latency, such as 48 hours, for awareness of privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to a private key compromised;
  AND
- A medium/blog link for sharing the time-lock contract and multi-signers addresses information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the deployed time-lock address.

- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.
- Provide a link to the **medium/blog** with all of the above information included.

Long Term:

A combination of a time-lock on the contract upgrade operation and a DAO for controlling the upgrade operation mitigate the contract upgrade risk by applying transparency and decentralization.

- A time-lock with reasonable latency, such as 48 hours, for community awareness of privileged operations;
  AND
- Introduction of a DAO, governance, or voting module to increase decentralization, transparency, and user involvement;
  AND
- A medium/blog link for sharing the time-lock contract, multi-signers addresses, and DAO information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the deployed time-lock address.
- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.
- Provide a link to the **medium/blog** with all of the above information included.

Permanent:

Renouncing ownership of the `admin` account or removing the upgrade functionality can *fully* resolve the risk.

- Renounce the ownership and never claim back the privileged role;
  OR
- Remove the risky functionality.

Note: we recommend the project team consider the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

Alleviation

[StakeMyGold, 04/01/2026]: The team acknowledged the issue and adopted the multi-sig solution at the current stage. The StakeMyGold has transferred the admin role to a Gnosis in transaction.

- Gnosis safe contract address: https://etherscan.io/address/0xf75FEa6Baa04bFdc2521fE0d428db0d128C26890#code
- The 2/3 multisign addresses:
  - 0x27cb9db3c65ae5e25c7ea8216008f1ff70ef43fa
  - 0x390e4447014818f43323efdb9b3fb47a24aba92a

- 0x664a03060920ae0e02d6b4d7c421b7910e2d01ee

STA-12 | Centralized Withdrawal Risk

Category	Severity	Location	Status
Centralization	**Centralization**	**contracts/src/StakedGGBR.sol (init-0409): 554~570**	**2/3 Multi-Sig**

Description

The `ADMIN_ROLE` is allowed to withdraw the underlying asset token via `emergencyWithdraw()`. Any compromise of the `ADMIN_ROLE` account may allow an attacker to drain all asset tokens from the contract, causing the asset value per share to drop sharply, or even making it impossible for users to withdraw their assets.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND

- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[StakeMyGold, 04/30/2026]: The StakedGGBR has transferred the `ADMIN_ROLE` to a Gnosis in transaction, the `DEFAULT_ADMIN_ROLE` to the same Gnosis in transaction.

- Gnosis safe contract address: https://etherscan.io/address/0x51820e72ba73fdab0e4471d1a2b20cb9bf6ee658
- The 2/3 multisign addresses:
  - 0x27cb9db3c65ae5e25c7ea8216008f1ff70ef43fa
  - 0x390e4447014818f43323efdb9b3fb47a24aba92a
  - 0x664a03060920ae0e02d6b4d7c421b7910e2d01ee

STA-01 Rewards Continuing Beyond Lock Period Leads To Unbounded Reward Drain

Category	Severity	Location	Status
Design Issue	Medium	contracts/StakeMyGold.sol (init): 357~367	Resolved

Description

The unstake function settles a position by calling `_calculateStakeRewards`, which computes rewards from `stakeEntry.startTime` all the way to the current `block.timestamp`. The issue is that the calculation never caps timeElapsed at `stakeEntry.lockPeriod`, so a stake created for 90 days, 183 days, or 365 days keeps earning forever as long as the user delays unstake. As a result, users can treat a fixed-term pool as a perpetual yield position and continuously extract rewards beyond the product's intended duration, which can gradually deplete the protocol's reward reserve.

```
357       */
358      function unstake(uint256 stakeId) external nonReentrant whenNotPaused  {
359          UserStakes storage userStakesData = userStakes[msg.sender];
360          if (!userStakesData.hasActiveStakes) revert NotStaking();
361
362          StakeEntry storage stakeEntry = userStakesData.stakes[stakeId];
363          if (stakeEntry.id == 0) revert StakeNotFound();
364          if (!stakeEntry.active) revert StakeNotActive();
365
366          // Calculate rewards for this specific stake
367          (uint256 stakeRewards, uint256 rehypothecationProfit) =
_calculateStakeRewards(stakeEntry);
```

Scenario

1. Alice stakes into the 90-day pool with the expectation that rewards are only meant to accrue for 90 days.
2. She does not call unstake when the lock period ends and instead waits another 180 days.
3. When Alice finally calls unstake, `_calculateStakeRewards` pays her for the entire elapsed time rather than only the original 90-day term.
4. If many users behave the same way, the protocol distributes materially more rewards than it budgeted for these fixed-term products.

Recommendation

Cap the reward accrual window at `stakeEntry.lockPeriod`, for example by using `min(block.timestamp, stakeEntry.startTime + stakeEntry.lockPeriod)` as the reward end time. If the intended design is to support post-maturity yield, document it explicitly and account for the additional liability in reserve management.

Alleviation

[StakeMyGold, 04/03/2026]: The team heeded the advice and resolved the issue by limiting the stake time in commit 954455b1c3b129fa40036b8fb5441d952d730b30

STA-02 Rehypothecation Rate Reduction Leads To Permanent Unstake Reverts

Category	Severity	Location	Status
Design Issue	Medium	contracts/StakeMyGold.sol (init): 451~456	Resolved

Description

The `_calculateStakeRewards` function calculates user rewards with the stake's fixed `periodDailyRate`, then derives `rehypothecationProfit` as the rehypothecation return minus those rewards. The issue is that `REHYPO_DAILY_RATE` is global and admin-configurable, while existing stakes keep their original `periodDailyRate`; if `REHYPO_APR` is later lowered below the APR promised to any active stake, the subtraction in `rehypothecationProfit` becomes negative and underflows. Because Solidity 0.8 reverts on underflow, unstake will revert as well, which can trap user funds and prevent affected users from exiting.

```
451     function _calculateStakeRewards(StakeEntry storage stakeEntry) internal
view returns (uint256 rewards, uint256 rehypothecationProfit) {
452         uint256 timeElapsed = block.timestamp - stakeEntry.startTime;
453         if (timeElapsed > 0) {
454             // Calculate daily rewards using period-specific rate
455             rewards = (stakeEntry.principal * stakeEntry.periodDailyRate *
 timeElapsed) / (RATE_PRECISION * 1 days);
456             rehypothecationProfit = (stakeEntry.principal * REHYPO_DAILY_RATE *
 timeElapsed) / (RATE_PRECISION * 1 days) - rewards;
```

Scenario

1. Bob stakes in a pool that locks in an 12% APR.
2. Later, the admin calls setRehypothecationRate and reduces `REHYPO_APR` to 8%.
3. When Bob tries to call unstake, `_calculateStakeRewards` computes a negative `rehypothecationProfit` and the transaction reverts.
4. Bob cannot withdraw his principal or rewards until the admin raises the rehypothecation rate again or the contract is upgraded.

Recommendation

Do not derive a user's withdrawal path from a subtraction that can go negative. A safer approach is to cap `rehypothecationProfit` at zero, or track protocol profit independently from user reward settlement. You should also prevent `setRehypothecationRate` from being set below the maximum APR of active stakes, or otherwise guarantee that changes cannot break withdrawals for existing positions.

Alleviation

[StakeMyGold, 04/03/2026]: The team heeded the advice and resolved the issue by replacing the direct subtraction with a bounded calculation that floors rehypothecationProfit at 0 when rehypothecation yield is below user rewards, preventing underflow and unstake reverts, in commit 954455b1c3b129fa40036b8fb5441d952d730b30

STA-03 | User Principal Can Be Used To Fund Other Users' Rewards

Category	Severity	Location	Status
Design Issue	Medium	contracts/StakeMyGold.sol (init): 392	Acknowledged

Description

The contract does not segregate staked principal from reward reserves. All `ggbrToken` held by the contract sits in a single pooled balance, and `unstake()` pays `principal + finalRewards` directly from that shared balance. As a result, newly deposited user principal can economically subsidize earlier users' reward payouts whenever external yield or prefunded reserves are insufficient.

User deposits are transferred into the contract's single `ggbrToken` balance:

```
ggbrToken.safeTransferFrom(msg.sender, address(this), amount);
...
totalStaked += amount;
```

Later, withdrawals are paid from that same pooled balance:

```
ggbrToken.safeTransfer(msg.sender, principal + finalRewards);
totalRewardsPaid += finalRewards;
```

There is no onchain reserve segregation such as:

- principal-only balance tracking,
- reward-only balance tracking,
- or a solvency check proving rewards are backed by realized profit before payout.

Instead, rewards are computed and paid as long as the contract's current token balance is sufficient. This means the source of payout is effectively:

```
shared contract balance = user principal + prefunded tokens + returned ION funds +
any other inbound tokens
```

The contract also treats rehypothecation profit as a formula-derived value rather than realized cash flow:

```
rehypothecationProfit =
    (stakeEntry.principal * REHYPO_DAILY_RATE * timeElapsed) / (RATE_PRECISION * 1
days)
    - rewards;
```

So if actual external yield is insufficient or delayed, the system can still pay early users' rewards out of whatever tokens are currently sitting in the contract, including principal recently deposited by later users.

This creates a commingled-balance model where:

1. user A stakes,
2. user B stakes later,
3. user A unstakes and receives rewards,
4. part of user A's reward may be sourced from user B's deposited principal rather than realized protocol profit.

Recommendation

Separate principal accounting from reward funding and only allow reward payouts from explicitly backed sources.

Alleviation

[StakeMyGold, 04/03/2026]: The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

> The yield is deposited by the yield provider I-ON Digital - a US publicly traded company. This is managed through their Fireblocks custody.

STA-11 Short-Term Pool Design Leads To Meaningless Product Segmentation

Category	Severity	Location	Status
Design Issue, Financial Manipulation	Medium	contracts/StakeMyGold.sol (init): 153~159	Acknowledged

Description

The `initialize` function configures the default staking products and the early-withdrawal penalty model. The issue is that the current parameters make the 365-day pool economically dominate the 90-day pool, because users can enter the higher-yield 365-day option and still exit early while only losing 30% of accrued rewards. This means the shorter pool provides little practical value, distorts user incentives, and can cause most rational users to cluster into the longest-duration product instead of selecting lock periods that match the intended risk and liquidity profile.

```
153         _addPeriod(90, 800);   // 90 days, 8% APR
154         _addPeriod(183, 1000);  // 183 days, 10% APR
155         _addPeriod(365, 1200);  // 365 days, 12% APR
156
157         // Initialize penalty rate
158         PENALTY_RATE = 3000; // 30% penalty rate in basis points
159         REHYPO_APR = 1300; // 13% rehypothecation rate in basis points
```

Scenario

1. Alice wants exposure for about 90 days and compares the available pools.
2. She notices the 90-day pool only offers 8% APR, while the 365-day pool offers 12% APR.
3. Alice chooses the 365-day pool, stays for roughly 90 days, and then exits early, losing only 30% of the rewards she earned during that period.
4. Even after the penalty, Alice can end up with economics that are comparable to or better than the dedicated 90-day pool, so the shorter product becomes unattractive.

Recommendation

Rebalance the product design so shorter lock periods remain meaningfully differentiated. This can be done by increasing the early-exit penalty, using principal haircuts for premature exits, reducing the APR gap structure, or making long-duration rewards vest only after maturity. The key goal is to ensure users cannot replicate or outperform short-term pools simply by entering a long-term pool and exiting early.

Alleviation

[StakeMyGold, 04/03/2026]: The team acknowledged the issue and decided not to implement the recommended change in the current engagement. Moreover, the team will change current APR values for periods in the future.

STA-13 Withdrawal Request Period Withdrawn Amount And Shares Are Still Included In Calculations

Category	Severity	Location	Status
Design Issue	Medium	contracts/src/StakedGGBR.sol (init-0409): 313~322, 330~339, 389~421	Resolved

Description

When a withdrawal request is created via `requestRedeem()` or `requestWithdraw()`, the shares and assets pending withdrawal are not excluded from `totalAsset()` and `totalSupply()`.

`StakedGGBR.sol`

```
389     function _createWithdrawalRequest(
390         address owner,
391         address receiver,
392         uint256 shares,
393         uint256 assets
394     ) internal returns (uint256 requestId) {
395         // Check user has enough shares (accounting for pending withdrawals)
396         uint256 availableShares = _getAvailableShares(owner);
397         if (shares > availableShares) {
398             revert InsufficientShares(shares, availableShares);
399         }
400 
401         requestId = nextWithdrawalId++;
402         uint256 claimableTime = block.timestamp + withdrawalDelay;
403 
404         withdrawalRequests[requestId] = WithdrawalRequest({
405             owner: owner,
406             receiver: receiver,
407             shares: shares,
408             assets: assets,
409             requestTime: block.timestamp,
410             claimableTime: claimableTime,
411             claimed: false,
412             cancelled: false
413         });
414 
415         _userWithdrawalIds[owner].push(requestId);
416 
417         // Transfer shares to contract (locked until claim/cancel)
418         _transfer(owner, address(this), shares);
419 
420         emit WithdrawalRequested(requestId, owner, receiver, shares, assets,
claimableTime);
421     }
```

`ERC4626.sol

```
    function totalAssets() public view virtual returns (uint256) {
        return IERC20(asset()).balanceOf(address(this));
    }
```

`ERC20.sol

```
    function totalSupply() public view virtual returns (uint256) {
        return _totalSupply;
    }
```

As a result, these shares and assets continue to be included in subsequent calculations before the actual claim:

`ERC4626.sol

```
    function _convertToShares(uint256 assets, Math.Rounding rounding) internal view
virtual returns (uint256) {
        return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(), totalAssets()
+ 1, rounding);
    }
```

`ERC4626.sol

```
    function _convertToAssets(uint256 shares, Math.Rounding rounding) internal view
virtual returns (uint256) {
        return shares.mulDiv(totalAssets() + 1, totalSupply() + 10 **
_decimalsOffset(), rounding);
    }
```

Moreover, when `claimWithdrawal()` is eventually executed, it is still affected by the then-current shares and assets, leading to an unexpected amount of assets being received.

`StakedGGBR.sol`

```
 389     function claimWithdrawal(uint256 requestId) external nonReentrant returns (
uint256 assets) {
 390         WithdrawalRequest storage request = withdrawalRequests[requestId];
 391
 392         if (request.claimed) revert WithdrawalAlreadyClaimed(requestId);
 393         if (request.cancelled) revert WithdrawalAlreadyCancelled(requestId);
 394         if (block.timestamp < request.claimableTime) {
 395             revert WithdrawalNotClaimable(requestId, request.claimableTime);
 396         }
 397
 398         request.claimed = true;
 399
 400  @>     assets = _convertToAssets(request.shares, Math.Rounding.Floor);
 401  @>     request.assets = assets;
// Store actual amount paid for off-chain consumers
 402
 403         // Burn shares and transfer assets
 404         _burn(address(this), request.shares);
 405  @>     IERC20(asset()).safeTransfer(request.receiver, assets);
 406
 407         emit WithdrawalClaimed(requestId, request.receiver, assets);
 408     }
```

This design can lead to several issues:

1. The assets corresponding to the pending shares remain affected during the request period, meaning that yield from the `YieldController` can still accrue. In this case, a user who deposits and immediately submits a withdrawal request can withdraw their funds at any time after the withdrawal delay, while still retaining yield that should not have been earned after withdrawal.

2. The assets received during `claimWithdrawal()` may differ from those at the time the withdrawal request was created.
3. When someone calls `claimWithdrawal()` and withdraws assets, it affects the current `totalAsset()` and `totalSupply()`, thereby impacting the amount of assets that other users with active withdrawal requests will receive after `claimWithdrawal()`.

Recommendation

We recommend introducing `pendingShares` and `pendingAssets` to track all shares and assets that are pending during the withdrawal request period. Additionally, `totalAsset()` and `totalSupply()` should be overwrited to exclude these amounts. Furthermore, during `claimWithdrawal()`, users should receive the previously recorded asset amount, and the corresponding `pendingShares` and `pendingAssets` should be reduced, rather than recalculating the asset amount. Similarly, `cancelWithdrawal()` should deduct the corresponding amounts from `pendingShares` and `pendingAssets`.

Alleviation

[StakeMyGold, 04/23/2026]: The team heeded the advice and resolved the issue in commit fd2cbf743a4f7d036647b18083563caa6f632504

STA-04 | `stake()` Is Incompatible With Fee-On-Transfer Tokens

Category	Severity	Location	Status
Coding Issue	Minor	contracts/StakeMyGold.sol (init): 317~334	Resolved

Description

The staking logic credits users with the requested amount, not the actual number of tokens received by the contract. If the staking token charges transfer fees or otherwise delivers less than amount, the contract will over-credit principal and totalStaked. This creates an accounting mismatch that can later cause insolvency or withdrawal failure.

The current staking flow assumes nominal transfer amount equals received amount:

```
311  function stake(uint256 amount, uint256 lockDays) external
 nonReentrant whenNotPaused {
312      if (amount == 0) revert InvalidAmount();
313      if (amount < minStakeAmount) revert BelowMinimumStake();
314      if (maxStakeAmount > 0 && amount > maxStakeAmount) revert AboveMaximumStake
();
315      if (!periodConfigs[lockDays].exists) revert PeriodNotExists();
316
317      ggbrToken.safeTransferFrom(msg.sender, address(this), amount);
318
319      PeriodConfig memory periodConfig = periodConfigs[lockDays];
320
321      uint256 stakeId = nextStakeId++;
322      StakeEntry memory newStake = StakeEntry({
323          id: stakeId,
324          principal: amount,
325          startTime: block.timestamp,
326          lockPeriod: lockDays * 1 days,
327          lockDays: lockDays,
328          periodAPR: periodConfig.apr,
329          periodDailyRate: periodConfig.dailyRate,
330          active: true
331      });
332
333      userStakes[msg.sender].stakes[stakeId] = newStake;
334      totalStaked += amount;
335  }
```

For a fee-on-transfer token, this assumption is false.

Example:

1. User calls stake(100e18, 365).
2. Token charges a 2% transfer fee.

3. Contract receives only 98e18.
4. The contract still records:

```
principal: 100e18
totalStaked += 100e18
```

That means the protocol has created a 100e18 principal liability while only receiving 98e18 of actual assets.

Later, withdrawal pays against the recorded principal, not the received amount:

```
ggbrToken.safeTransfer(msg.sender, principal + finalRewards);
```

This creates a direct accounting gap:

```
recorded principal > actual assets received
```

Over time, repeated deposits of fee-on-transfer tokens can make the pool insolvent even if all code paths execute exactly as written.

Recommendation

Use balance-delta accounting on deposits and stake based on the actual amount received.

Alleviation

[StakeMyGold, 04/03/2026]: The team heeded the advice and resolved the issue in the latest commit 954455b1c3b129fa40036b8fb5441d952d730b30.

STA-05 `depositFromION()` Does Not Restrict Repayments To `ionWallet`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/StakeMyGold.sol (init): 501~510	Resolved

Description

`depositFromION(uint256 amount)` is documented as the repayment path for the configured `ionWallet`, but the implementation never verifies `msg.sender == ionWallet`. As a result, any address with sufficient `ggbrToken` balance and allowance can transfer its own tokens into the contract and reduce `totalPaidToIONWallet`. This corrupts the accounting semantics of outstanding ION receivables and can cause later legitimate ION repayments to revert if they exceed the remaining recorded balance.

The inline comments describes `depositFromION()` as an ION-specific settlement path:

```
498  @dev Only callable by the ION wallet.
Increases pendingIONTransfer by the received amount.
```

However, the implementation does not enforce that requirement:

```
501  function depositFromION(uint256 amount) external nonReentrant whenNotPaused {
502          if (amount == 0) revert InvalidAmount();
503
504          // Transfer GGBRO tokens from ION wallet to this contract
505          ggbrToken.safeTransferFrom(msg.sender, address(this), amount);
506
507          totalPaidToIONWallet -= amount;
508
509          emit PrincipalDeposited(msg.sender, amount, block.timestamp);
510      }
```

As written, any address can:

1. hold ggbrToken,
2. approve the staking contract,
3. call depositFromION(amount),
4. and reduce:

```
totalPaidToIONWallet -= amount;
```

This breaks the accounting semantics of `totalPaidToIONWallet`. Observers can no longer rely on it as a trustworthy

measure of what ION still owes.

This issue does not directly let an attacker steal funds, since the caller must transfer their own tokens into the contract. The impact is on settlement correctness, accounting integrity, and potential repayment liveness.

Recommendation

Restrict `depositFromION()` to the configured `ionWallet` and preserve the semantic meaning of `totalPaidToIONWallet`.

Alleviation

[StakeMyGold, 04/03/2026]: The team heeded the advice and resolved the issue in the latest commit 954455b1c3b129fa40036b8fb5441d952d730b30.

STA-06 Unbounded Per-User Stake Growth Can Cause Withdrawal DoS

Category	Severity	Location	Status
Denial of Service	Minor	contracts/StakeMyGold.sol (init): 466~478	Resolved

Description

The contract stores each user's active stake IDs in a dynamic array and removes entries by linear search. As the number of active stakes for a single user grows, both `unstake()` and summary queries become increasingly expensive. A user with a sufficiently fragmented position set may eventually face failed or impractical withdrawals due to gas growth, creating a self-DoS risk.

Each user's active positions are tracked in an array:

```
struct UserStakes {
    uint256[] activeStakeIds;
    mapping(uint256 => StakeEntry) stakes;
    bool hasActiveStakes;
}
```

Whenever a user stakes, the new stakeId is appended:

```
userStakesData.stakes[stakeId] = newStake;
userStakesData.activeStakeIds.push(stakeId);
userStakesData.hasActiveStakes = true;
```

When the user later unstakes, the contract removes the ID by scanning the full array until it finds the target:

```
function _removeStakeFromActiveArray(address user, uint256 stakeId) internal {
    UserStakes storage userStakesData = userStakes[user];
    uint256[] storage activeIds = userStakesData.activeStakeIds;

    for (uint256 i = 0; i < activeIds.length; i++) {
        if (activeIds[i] == stakeId) {
            activeIds[i] = activeIds[activeIds.length - 1];
            activeIds.pop();
            break;
        }
    }
}
```

This makes removal cost O(n) in the number of active stakes for that user. The same pattern appears in getUserStakesSummary(), which iterates over all active stake IDs. There is no cap on how many active stake entries a single user may create.

a user can fragment their position across many active stakes. Over time, this can make:

- `unstake()` increasingly expensive,
- summary queries increasingly slow,
- and eventual withdrawals operationally difficult or non-executable under realistic gas limits.

Recommendation

Consider imposing a maximum number of active stakes per user.

Alleviation

[StakeMyGold, 04/03/2026]: The team heeded the advice and resolved the issue in the latest commit 954455b1c3b129fa40036b8fb5441d952d730b30.

STA-07 `getPlatformStats()` Can Revert Due To Underflow In Reserve Calculation

Category	Severity	Location	Status
Logical Issue	Minor	contracts/StakeMyGold.sol (init): 664	Resolved

Description

`getPlatformStats()` attempts to compute a reserve-style balance using `balance`, `totalPaidToIONWallet`, and `totalStaked`, but the formula is algebraically unsafe. The outer condition checks `balance + totalPaidToIONWallet >= totalStaked`, yet the selected branch still evaluates `totalStaked - totalPaidToIONWallet`. If `totalPaidToIONWallet > totalStaked`, that inner subtraction underflows and the entire view function reverts.

The current implementation is:

```
function getPlatformStats() external view returns (
    uint256 _totalStaked,
    uint256 _totalRewardsPaid,
    uint256 _totalPenaltiesCollected,
    uint256 _activeStakerCount,
    uint256 _totalPaidToIONWallet,
    uint256 _totalProfitFromRehypothecation,
    uint256 _remainingBalanceFromReserve
) {
    uint256 balance = ggbrToken.balanceOf(address(this));
    uint256 remainingBalanceFromReserve =
        (balance + totalPaidToIONWallet >= totalStaked)
            ? balance - (totalStaked - totalPaidToIONWallet)
            : 0;

    return (
        totalStaked,
        totalRewardsPaid,
        totalPenaltiesCollected,
        stakers.length,
        totalPaidToIONWallet,
        totalProfitFromRehypothecation,
        remainingBalanceFromReserve
    );
}
```

The issue is that the branch condition does not guarantee the safety of the inner subtraction:

```
totalStaked - totalPaidToIONWallet
```

Example:

```
balance = 100
totalStaked = 50
totalPaidToIONWallet = 80
```

The condition passes:

```
balance + totalPaidToIONWallet = 180 >= 50
```

but the branch computes:

```
balance - (totalStaked - totalPaidToIONWallet)
= 100 - (50 - 80)
```

The inner `(50-80)` underflows under Solidity ^0.8.20, causing the entire function to revert.

Recommendation

Rewrite the reserve formula using the guarded addition form instead of nested subtraction.

Safer pattern:

```
uint256 remainingBalanceFromReserve =
    (balance + totalPaidToIONWallet >= totalStaked)
        ? balance + totalPaidToIONWallet - totalStaked
        : 0;
```

Alleviation

[StakeMyGold, 04/03/2026]: The team heeded the advice and resolved the issue in the latest commit 954455b1c3b129fa40036b8fb5441d952d730b30.

STA-14 Reward Dilution Caused By Non-Linear Time-Based Distribution Enabling Front-Running Of `depositYield()`

Category	Severity	Location	Status
Design Issue	Minor	contracts/src/StakedGGBR.sol (init-0409): 439~465	Acknowledged

Description

In `StakedGGBR`, rewards are added via the `YieldController` calling `depositYield()`, which increases rewards instantaneously rather than distributing them linearly over time. If a user front-runs `depositYield()` and deposits assets into the contract, they can immediately dilute others' rewards and receive a share of the rewards without accruing them over time. This is unfair to users who have been staking for a longer period.

Although the contract includes a `withdrawalDelay` to prevent instant withdrawals, and `injectYield()` in the `YieldController` imposes an upper limit that partially mitigates the issue, the problem still persists.

Recommendation

We recommend adopting a time-based reward distribution model similar to designs used in Synthetix or MasterChef to prevent users from instantly capturing a disproportionate amount of rewards.

Alleviation

[StakeMyGold, 04/21/2026]: Acknowledged (Only Admin will deposit yield, so users can't front run `depositYield()`.)

[CertiK, 04/21/2026]: Thank you for your response. What we mean is that a user can front-run by calling `deposit()` before the official `depositYield()` is executed. This allows them to immediately share in the rewards, which is unfair to other long-term stakers.

[StakeMyGold, 04/23/2026]: We acknowledge and accept this finding as a known economic fairness risk (not an access-control issue).

STA-15 | Missing Validity Checks In `claimWithdrawal()`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/src/StakedGGBR.sol (init-0409): 346~365	Resolved

Description

`claimWithdrawal()` does not validate whether the input `requestId` is a valid ID. When an invalid ID is provided, the function can still execute and complete the transaction successfully.

`StakedGGBR.sol`

```
346     function claimWithdrawal(uint256 requestId) external nonReentrant returns (
uint256 assets) {
347         WithdrawalRequest storage request = withdrawalRequests[requestId];
348
349         if (request.claimed) revert WithdrawalAlreadyClaimed(requestId);
350         if (request.cancelled) revert WithdrawalAlreadyCancelled(requestId);
351         if (block.timestamp < request.claimableTime) {
352             revert WithdrawalNotClaimable(requestId, request.claimableTime);
353         }
354
355         request.claimed = true;
356
357         assets = _convertToAssets(request.shares, Math.Rounding.Floor);
358         request.assets = assets;
// Store actual amount paid for off-chain consumers
359
360         // Burn shares and transfer assets
361         _burn(address(this), request.shares);
362         IERC20(asset()).safeTransfer(request.receiver, assets);
363
364         emit WithdrawalClaimed(requestId, request.receiver, assets);
365     }
```

Recommendation

We recommend adding a check in `claimWithdrawal()` to ensure that `requestId` is less than `nextWithdrawalId`.

Alleviation

[StakeMyGold, 04/23/2026]: The team heeded the advice and resolved the issue in commit fd2cbf743a4f7d036647b18083563caa6f632504

STA-16 | Direct Fund Injection Via Transfer

Category	Severity	Location	Status
Logical Issue	Minor	contracts/src/StakedGGBR.sol (init-0409): 439~465; contracts/src/YieldController.sol (init-0409): 112~116	Acknowledged

Description

The contract is expected to inject funds into `StakedGGBR` through `YieldController.injectYield()` → `StakedGGBR.depositYield()`. However, directly transferring tokens to `StakedGGBR` achieves the same effect because `totalAssets()` is implemented as the contract's `balanceOf`: **`ERC4626.sol`**

```
139     function totalAssets() public view virtual returns (uint256) {
140         return IERC20(asset()).balanceOf(address(this));
141     }
```

This allows direct transfers to bypass the checks and accounting performed in `YieldController.injectYield()` and `StakedGGBR.depositYield()`, rendering their validation and recording mechanisms ineffective.

Recommendation

We would like to confirm whether this is intended behavior.

Alleviation

[StakeMyGold, 04/23/2026]: Acknowledged (This is intended)

STA-20 | Issue Of Contract Yield Being Absorbed By Virtual Shares

Category	Severity	Location	Status
Logical Issue	Minor	contracts/src/StakedGGBR.sol (init-0409): 439~465	Resolved

Description

`depositYield()` does not prevent being called when `totalSupply()` of `StakedGGBR` is zero. If it is invoked while `totalSupply() == 0`, the injected yield will be absorbed by the virtual shares defined by `10 ** _decimalsOffset()`.

Additionally, virtual shares also receive a portion of the yield distributed via `depositYield()` under normal conditions. We would like to confirm with the team whether this is an intended design.

Recommendation

We would like to confirm with the team whether this is an intended design. In addition, the project team should hold a portion of StakedGGBR to prevent a significant portion of the yield from being absorbed by virtual shares.

Alleviation

[StakeMyGold, 04/25/2026]: Confirmed intended design for non-zero supply.

STA-21 Dilution Of Existing Rewards When Funds Re-Enter The Pool Via `cancelWithdrawal()`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/src/StakedGGBR.sol (fix-0421): 406~423	Resolved

Description

When a user submits a withdrawal request and later changes their mind, calling `cancelWithdrawal()` will return their vault shares to them as-is. However, during the pending withdrawal period, `depositYield()` may have been called to distribute rewards. As a result, the value of each share at the time of cancellation may be higher than when the withdrawal request was initially made, meaning the same number of shares now represents a larger amount of assets.

In this case, by canceling the withdrawal, the user effectively receives a portion of the rewards distributed during the pending period—rewards they should not be entitled to.

Recommendation

We recommend that, upon calling `cancelWithdrawal()`, the contract recalculates the number of shares corresponding to the user's withdrawal request asset amount and returns only that amount of shares to the user, while burning any excess shares.

Alleviation

[StakeMyGold, 04/27/2026]: The team heeded the advice and resolved the issue in commit 4b0a2b0bca93fc102347580305e47668947b7246

STA-08 | Unused Variables

Category	Severity	Location	Status
Coding Issue	Informational	contracts/StakeMyGold.sol (init): 92, 96, 100~101, 113, 114, 117	Resolved

Description

The following events are unused:

```
event RewardsClaimed(address indexed user, uint256 amount);
event TokenFactoryUpdated(address oldFactory, address newFactory);
event StakerRoleGranted(address indexed account);
event StakerRoleRevoked(address indexed account);
```

The following errors are unused:

```
error NoRewardsToClaim();
error InvalidLockPeriod();
error LockPeriodNotEnded();
```

Recommendation

Consider removing unused variables.

Alleviation

[StakeMyGold, 04/03/2026]: The team heeded the advice and resolved the issue in the latest commit 954455b1c3b129fa40036b8fb5441d952d730b30.

STA-17 | Unit Inconsistency In `exchangeRate()` Calculation

Category	Severity	Location	Status
Inconsistency	Informational	contracts/src/StakedGGBR.sol (init-0409): 578~580	Resolved

Description

`exchangeRate()` uses `1e18` for share conversion.

`StakedGGBR.sol`

```
578     function exchangeRate() public view returns (uint256) {
579         return _convertToAssets(1e18, Math.Rounding.Floor);
580     }
```

However, the actual share decimals are `_underlyingDecimals + _decimalsOffset()`, which equals 24.

`ERC4626.sol`

```
129     function decimals() public view virtual override(IERC20Metadata, ERC20)
returns (uint8) {
130         return _underlyingDecimals + _decimalsOffset();
131     }
```

This design results in a rate with inconsistent units.

Recommendation

We would like to confirm whether this is the intended behavior. If not, it should be updated to use the correct decimals, i.e., `1e24`

Alleviation

[StakeMyGold, 04/23/2026]: The client confirmed that this was not an intentional design and resolved the issue by updating the unit in `exchangeRate()` to `10 ** decimals()` in commit fd2cbf743a4f7d036647b18083563caa6f632504

STA-18 | `StakedGGBR` Interface ID Not Included In `supportsInterface`

Category	Severity	Location	Status
Design Issue	Informational	contracts/src/StakedGGBR.sol (init-0409): 679~688	Resolved

Description

`suppportsInterface` does not include the `StakedGGBR` interface ID for validation, and instead directly returns the result from the inherited `AccessControl` contract.

Recommendation

We recommend adding the `StakedGGBR` interface ID to the function.

Alleviation

[StakeMyGold, 04/23/2026]: The team heeded the advice and resolved the issue in commit fd2cbf743a4f7d036647b18083563caa6f632504

OPTIMIZATIONS | STAKEMYGOLD

ID	Title	Category	Severity	Status
STA-19	Excessive Gas Consumption In `getPendingWithdrawals()` When The Array Is Too Large	Gas Optimization	Optimization	Resolved

STA-19 Excessive Gas Consumption In `getPendingWithdrawals()` When The Array Is Too Large

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/src/StakedGGBR.sol (init-0409): 622~652	Resolved

Description

`getPendingWithdrawals()` iterates over the entire `_userWithdrawalIds[user]` array. This array grows without restriction and does not have a `pop` mechanism. When it becomes too large, calling this function on-chain can consume excessive gas.

`StakedGGBR.sol`

```
622     function get√(address user) external view returns (
623         uint256[] memory requestIds,
624         uint256 totalPendingShares,
625         uint256 totalPendingAssets
626     ) {
627 @>      uint256[] storage allIds = _userWithdrawalIds[user];
628         uint256 pendingCount;
629
630         // Count pending requests
631 @>      for (uint256 i; i < allIds.length;) {
632             WithdrawalRequest storage request = withdrawalRequests[allIds[i]];
633             if (!request.claimed && !request.cancelled) {
634                 pendingCount++;
635             }
636             unchecked { ++i; }
637         }
638
639         // Build result arrays
640 @>      requestIds = new uint256[](pendingCount);
641         uint256 idx;
642         for (uint256 i; i < allIds.length;) {
643             WithdrawalRequest storage request = withdrawalRequests[allIds[i]];
644             if (!request.claimed && !request.cancelled) {
645                 requestIds[idx] = allIds[i];
646                 totalPendingShares += request.shares;
647                 totalPendingAssets += request.assets;
648                 idx++;
649             }
650             unchecked { ++i; }
651         }
652     }
653     ```
```

Recommendation

We recommend maintaining a separate record of pending withdrawals and returning it directly when needed.

Alleviation

[StakeMyGold, 04/23/2026]: The team heeded the advice and resolved the issue in commit fd2cbf743a4f7d036647b18083563caa6f632504.

FORMAL VERIFICATION | STAKEMYGOLD

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied formal verification to prove that important functions in the smart contracts adhere to their expected behaviors.

Considered Functions And Scope

Considered Functions And Scope

Verification of contracts derived from AccessControl v4.4

We verified properties of the public interface of contracts that provide an AccessControl-v4.4 compatible API. This involves:

- The `hasRole` function, which returns `true` if an account has been granted a specific `role`.
- The `getRoleAdmin` function, which returns the admin role that controls a specific `role`.
- The `grantRole` and `revokeRole` functions, which are used for granting a `role` to an account and revoking a `role` from an `account`, respectively.
- The `renounceRole` function, which allows the calling account to revoke a `role` from itself.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
accesscontrol-hasrole-change-state	`hasRole` Function Does Not Change State
accesscontrol-default-admin-role	AccessControl Default Admin Role Invariance
accesscontrol-revokerole-correct-role-revoking	`revokeRole` Correctly Revokes Role
accesscontrol-grantrole-correct-role-granting	`grantRole` Correctly Grants Role
accesscontrol-getroleadmin-change-state	`getRoleAdmin` Function Does Not Change State
accesscontrol-renouncerole-revert-not-sender	`renounceRole` Reverts When Caller Is Not the Confirmation Address
accesscontrol-getroleadmin-succeed-always	`getRoleAdmin` Function Always Succeeds
accesscontrol-renouncerole-succeed-role-renouncing	`renounceRole` Successfully Renounces Role
accesscontrol-hasrole-succeed-always	`hasRole` Function Always Succeeds

Verification of ERC-20 Compliance

We verified properties of the public interface of those token contracts that implement the ERC-20 interface. This covers

- Functions `transfer` and `transferFrom` that are widely used for token transfers,
- functions `approve` and `allowance` that enable the owner of an account to delegate a certain subset of her tokens to another account (i.e. to grant an allowance), and
- the functions `balanceOf` and `totalSupply`, which are verified to correctly reflect the internal state of the contract.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
erc20-transferfrom-revert-zero-argument	`transferFrom` Fails for Transfers with Zero Address Arguments
erc20-transfer-correct-amount	`transfer` Transfers the Correct Amount in Transfers
erc20-transfer-revert-zero	`transfer` Prevents Transfers to the Zero Address
erc20-transfer-exceed-balance	`transfer` Fails if Requested Amount Exceeds Available Balance
erc20-transferfrom-fail-exceed-balance	`transferFrom` Fails if the Requested Amount Exceeds the Available Balance
erc20-allowance-change-state	`allowance` Does Not Change the Contract's State
erc20-balanceof-change-state	`balanceOf` Does Not Change the Contract's State
erc20-totalsupply-change-state	`totalSupply` Does Not Change the Contract's State
erc20-transferfrom-correct-allowance	`transferFrom` Updated the Allowance Correctly
erc20-transferfrom-correct-amount	`transferFrom` Transfers the Correct Amount in Transfers
erc20-transferfrom-fail-recipient-overflow	`transferFrom` Prevents Overflows in the Recipient's Balance
erc20-transfer-recipient-overflow	`transfer` Prevents Overflows in the Recipient's Balance
erc20-approve-revert-zero	`approve` Prevents Approvals For the Zero Address
erc20-balanceof-succeed-always	`balanceOf` Always Succeeds
erc20-totalsupply-succeed-always	`totalSupply` Always Succeeds
erc20-transferfrom-fail-exceed-allowance	`transferFrom` Fails if the Requested Amount Exceeds the Available Allowance
erc20-allowance-correct-value	`allowance` Returns Correct Value

Property Name	Title
erc20-transferfrom-false	If `transferFrom` Returns `false`, the Contract's State Is Unchanged
erc20-transferfrom-never-return-false	`transferFrom` Never Returns `false`
erc20-approve-correct-amount	`approve` Updates the Approval Mapping Correctly
erc20-transfer-never-return-false	`transfer` Never Returns `false`
erc20-transfer-false	If `transfer` Returns `false`, the Contract State Is Not Changed
erc20-allowance-succeed-always	`allowance` Always Succeeds
erc20-balanceof-correct-value	`balanceOf` Returns the Correct Value
erc20-approve-succeed-normal	`approve` Succeeds for Valid Inputs
erc20-approve-never-return-false	`approve` Never Returns `false`
erc20-totalsupply-correct-value	`totalSupply` Returns the Value of the Corresponding State Variable
erc20-approve-false	If `approve` Returns `false`, the Contract's State Is Unchanged

Verification Results

Verification Results

Detailed Results For Contract YieldController (contracts/src/YieldController.sol) In Commit 4b0a2b0bca93fc102347580305e47668947b7246

Verification of contracts derived from AccessControl v4.4

Detailed Results for Function `hasRole`

Property Name	Final Result	Remarks
accesscontrol-hasrole-change-state	● True	
accesscontrol-hasrole-succeed-always	● True	

Detailed Results for Function `DEFAULT_ADMIN_ROLE`

Property Name	Final Result	Remarks
accesscontrol-default-admin-role	● True	

Detailed Results for Function `revokeRole`

Property Name	Final Result	Remarks
accesscontrol-revokerole-correct-role-revoking	True	

Detailed Results for Function `grantRole`

Property Name	Final Result	Remarks
accesscontrol-grantrole-correct-role-granting	True	

Detailed Results for Function `getRoleAdmin`

Property Name	Final Result	Remarks
accesscontrol-getroleadmin-change-state	True	
accesscontrol-getroleadmin-succeed-always	True	

Detailed Results for Function `renounceRole`

Property Name	Final Result	Remarks
accesscontrol-renouncerole-revert-not-sender	True	
accesscontrol-renouncerole-succeed-role-renouncing	True	

Detailed Results For Contract YieldController (contracts/src/YieldController.sol) In Commit fd2cbf743a4f7d036647b18083563caa6f632504

Verification of contracts derived from AccessControl v4.4

Detailed Results for Function `renounceRole`

Property Name	Final Result	Remarks
accesscontrol-renouncerole-revert-not-sender	True	
accesscontrol-renouncerole-succeed-role-renouncing	True	

Detailed Results for Function `hasRole`

Property Name	Final Result	Remarks
accesscontrol-hasrole-succeed-always	True	
accesscontrol-hasrole-change-state	True	

Detailed Results for Function `getRoleAdmin`

Property Name	Final Result	Remarks
accesscontrol-getroleadmin-succeed-always	True	
accesscontrol-getroleadmin-change-state	True	

Detailed Results for Function `DEFAULT_ADMIN_ROLE`

Property Name	Final Result	Remarks
accesscontrol-default-admin-role	True	

Detailed Results for Function `revokeRole`

Property Name	Final Result	Remarks
accesscontrol-revokerole-correct-role-revoking	True	

Detailed Results for Function `grantRole`

Property Name	Final Result	Remarks
accesscontrol-grantrole-correct-role-granting	True	

Detailed Results For Contract YieldController (contracts/src/YieldController.sol) In Commit 52246af2b5a158c5116b7e47d21cd8d33b5ca271

Verification of contracts derived from AccessControl v4.4

Detailed Results for Function `renounceRole`

Property Name	Final Result	Remarks
accesscontrol-renouncerole-revert-not-sender	True	
accesscontrol-renouncerole-succeed-role-renouncing	True	

Detailed Results for Function `hasRole`

Property Name	Final Result	Remarks
accesscontrol-hasrole-succeed-always	True	
accesscontrol-hasrole-change-state	True	

Detailed Results for Function `DEFAULT_ADMIN_ROLE`

Property Name	Final Result	Remarks
accesscontrol-default-admin-role	True	

Detailed Results for Function `getRoleAdmin`

Property Name	Final Result	Remarks
accesscontrol-getroleadmin-change-state	True	
accesscontrol-getroleadmin-succeed-always	True	

Detailed Results for Function `grantRole`

Property Name	Final Result	Remarks
accesscontrol-grantrole-correct-role-granting	True	

Detailed Results for Function `revokeRole`

Property Name	Final Result	Remarks
accesscontrol-revokerole-correct-role-revoking	True	

In the remainder of this section, we list all contracts where formal verification of at least one property was not successful. There are several reasons why this could happen:

- False: The property is violated by the project.
- Inconclusive: The proof engine cannot prove or disprove the property due to timeouts or exceptions.
- Inapplicable: The property does not apply to the project.

Detailed Results For Contract StakedGGBR (contracts/src/StakedGGBR.sol) In Commit 4b0a2b0bca93fc102347580305e47668947b7246

Verification of ERC-20 Compliance

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-revert-zero-argument	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-fail-recipient-overflow	Inconclusive	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-false	True	
erc20-transferfrom-never-return-false	True	

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-correct-amount	True	
erc20-transfer-revert-zero	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-recipient-overflow	Inconclusive	
erc20-transfer-never-return-false	True	
erc20-transfer-false	True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-change-state	True	
erc20-allowance-correct-value	True	
erc20-allowance-succeed-always	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-change-state	True	
erc20-balanceof-succeed-always	True	
erc20-balanceof-correct-value	True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-change-state	True	
erc20-totalsupply-succeed-always	True	
erc20-totalsupply-correct-value	True	

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-revert-zero	True	
erc20-approve-correct-amount	True	
erc20-approve-succeed-normal	True	
erc20-approve-never-return-false	True	
erc20-approve-false	True	

Detailed Results For Contract StakedGGBR (contracts/src/StakedGGBR.sol) In Commit fd2cbf743a4f7d036647b18083563caa6f632504

Verification of ERC-20 Compliance

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-exceed-balance	True	
erc20-transfer-revert-zero	True	
erc20-transfer-correct-amount	True	
erc20-transfer-recipient-overflow	Inconclusive	
erc20-transfer-false	True	
erc20-transfer-never-return-false	True	

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-fail-recipient-overflow	Inconclusive	
erc20-transferfrom-revert-zero-argument	True	
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-false	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-change-state	True	
erc20-balanceof-correct-value	True	
erc20-balanceof-succeed-always	True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-change-state	True	
erc20-allowance-succeed-always	True	
erc20-allowance-correct-value	True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-change-state	True	
erc20-totalsupply-correct-value	True	
erc20-totalsupply-succeed-always	True	

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-never-return-false	True	
erc20-approve-false	True	
erc20-approve-revert-zero	True	
erc20-approve-succeed-normal	True	
erc20-approve-correct-amount	True	

Detailed Results For Contract StakedGGBR (contracts/src/StakedGGBR.sol) In Commit 52246af2b5a158c5116b7e47d21cd8d33b5ca271

Verification of ERC-20 Compliance

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-fail-recipient-overflow	Inconclusive	
erc20-transferfrom-revert-zero-argument	True	
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-false	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-correct-allowance	True	

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-recipient-overflow	Inconclusive	
erc20-transfer-never-return-false	True	
erc20-transfer-revert-zero	True	
erc20-transfer-false	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-correct-amount	True	

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-succeed-normal	True	
erc20-approve-never-return-false	True	
erc20-approve-false	True	
erc20-approve-revert-zero	True	
erc20-approve-correct-amount	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-correct-value	True	
erc20-balanceof-succeed-always	True	
erc20-balanceof-change-state	True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-succeed-always	True	
erc20-totalsupply-correct-value	True	
erc20-totalsupply-change-state	True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-correct-value	True	
erc20-allowance-change-state	True	
erc20-allowance-succeed-always	True	

APPENDIX | STAKEMYGOLD

Finding Categories

Categories	Description
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.
Financial Manipulation	Financial Manipulation findings indicate issues in design that may lead to financial losses.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Gas Optimization	Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Details on Formal Verification

Some Solidity smart contracts from this project have been formally verified. Each such contract was compiled into a mathematical model that reflects all its possible behaviors with respect to the property. The model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

The following assumptions and simplifications apply to our model:

- Certain low-level calls and inline assembly are not supported and may lead to a contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for property specifications

All properties are expressed in a behavioral interface specification language that CertiK has developed for Solidity, which allows us to specify the behavior of each function in terms of the contract state and its parameters and return values, as well as contract properties that are maintained by every observable state transition. Observable state transitions occur when the contract's external interface is invoked and the invocation does not revert, and when the contract's Ether balance is changed by the EVM due to another contract's "self-destruct" invocation. The specification language has the usual Boolean connectives, as well as the operator `\old` (used to denote the state of a variable before a state transition), and several types of specification clause:

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates to reason about the validity of atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `requires [cond]` - the condition `cond`, which refers to a function's parameters, return values, and contract state variables, must hold when a function is invoked in order for it to exhibit a specified behavior.
- `ensures [cond]` - the condition `cond`, which refers to a function's parameters, return values, and both `\old` and current contract state variables, is guaranteed to hold when a function returns if the corresponding requires condition held when it was invoked.
- `invariant [cond]` - the condition `cond`, which refers only to contract state variables, is guaranteed to hold at every observable contract state.
- `constraint [cond]` - the condition `cond`, which refers to both `\old` and current contract state variables, is guaranteed to hold at every observable contract state except for the initial state after construction (because there is no previous state); constraints are used to restrict how contract state can change over time.

Description of the Analyzed AccessControl-v4.4 Properties

Properties related to function `hasRole`

accesscontrol-hasrole-change-state

The `hasRole` function must not change any state variables.

Specification:

```
assignable \nothing;
```

accesscontrol-hasrole-succeed-always

The `hasRole` function must always succeed, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `DEFAULT_ADMIN_ROLE`

accesscontrol-default-admin-role

The default admin role must be invariant, ensuring consistent access control management.

Specification:

```
invariant DEFAULT_ADMIN_ROLE() == 0x00;
```

Properties related to function `revokeRole`

accesscontrol-revokerole-correct-role-revoking

After execution, `revokeRole` must ensure the specified account no longer has the revoked role.

Specification:

```
ensures !hasRole(role, account);
```

Properties related to function `grantRole`

accesscontrol-grantrole-correct-role-granting

After execution, `grantRole` must ensure the specified account has the granted role.

Specification:

```
ensures hasRole(role, account);
```

Properties related to function `getRoleAdmin`

accesscontrol-getroleadmin-change-state

The `getRoleAdmin` function must not change any state variables.

Specification:

```
assignable \nothing;
```

accesscontrol-getroleadmin-succeed-always

The `getRoleAdmin` function must always succeed, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `renounceRole`

accesscontrol-renouncerole-revert-not-sender

The `renounceRole` function must revert if the caller is not the same as `account`.

Specification:

```
reverts_when account != msg.sender;
```

accesscontrol-renouncerole-succeed-role-renouncing

After execution, `renounceRole` must ensure the caller no longer has the renounced role.

Specification:

```
ensures !hasRole(role, account);
```

Description of the Analyzed ERC-20 Properties

Properties related to function `transferFrom`

erc20-transferfrom-correct-allowance

All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` must decrease the allowance for address `msg.sender` over address `from` by the value in `amount`.

Specification:

```
ensures \result ==> allowance(\old(sender), msg.sender) == \old(allowance(sender,
msg.sender)) - \old(amount)
                  || (allowance(\old(sender), msg.sender) == \old(allowance(sender,
msg.sender)) && \old(allowance(sender, msg.sender)) == type(uint256).max);
```

erc20-transferfrom-correct-amount

All invocations of `transferFrom(from, dest, amount)` that succeed and that return `true` subtract the value in `amount` from the balance of address `from` and add the same value to the balance of address `dest`.

Specification:

```
requires recipient != sender;
requires balanceOf(recipient) + amount <= type(uint256).max;
ensures \result ==> balanceOf(\old(recipient)) == \old(balanceOf(recipient) +
amount)
                 && balanceOf(\old(sender)) == \old(balanceOf(sender) - amount);
  also
requires recipient == sender;
ensures \result ==> balanceOf(\old(recipient)) == \old(balanceOf(recipient));
```

erc20-transferfrom-fail-exceed-allowance

Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the allowance of address `msg.sender` must fail.

Specification:

```
requires msg.sender != sender;
requires amount > allowance(sender, msg.sender);
ensures !\result;
```

erc20-transferfrom-fail-exceed-balance

Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the balance of address `from` must fail.

Specification:

```
requires amount > balanceOf(sender);
ensures !\result;
```

erc20-transferfrom-fail-recipient-overflow

Any call of `transferFrom(from, dest, amount)` with a value in `amount` whose transfer would cause an overflow of the balance of address `dest` must fail.

Specification:

```
requires recipient != sender;
requires balanceOf(recipient) + amount > type(uint256).max;
ensures !\result;
```

erc20-transferfrom-false

If `transferFrom` returns `false` to signal a failure, it must undo all incurred state changes before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-transferfrom-never-return-false

The `transferFrom` function must never return `false`.

Specification:

```
ensures \result;
```

erc20-transferfrom-revert-zero-argument

All calls of the form `transferFrom(from, dest, amount)` must fail for transfers from or to the zero address.

Specification:

```
ensures \old(sender) == address(0) ==> !\result;
also
ensures \old(recipient) == address(0) ==> !\result;
```

Properties related to function `transfer`

erc20-transfer-correct-amount

All non-reverting invocations of `transfer(recipient, amount)` that return `true` must subtract the value in `amount` from the balance of `msg.sender` and add the same value to the balance of the `recipient` address.

Specification:

```
requires recipient != msg.sender;
requires balanceOf(recipient) + amount <= type(uint256).max;
ensures \result ==> balanceOf(recipient) == \old(balanceOf(recipient) + amount)
&& balanceOf(msg.sender) == \old(balanceOf(msg.sender) - amount);
  also
requires recipient == msg.sender;
ensures \result ==> balanceOf(msg.sender) == \old(balanceOf(msg.sender));
```

erc20-transfer-exceed-balance

Any transfer of an amount of tokens that exceeds the balance of `msg.sender` must fail.

Specification:

```
requires amount > balanceOf(msg.sender);
ensures !\result;
```

erc20-transfer-false

If the `transfer` function in contract `StakedGGBR` fails by returning `false`, it must undo all state changes it incurred before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-transfer-never-return-false

The transfer function must never return `false` to signal a failure.

Specification:

```
ensures \result;
```

erc20-transfer-recipient-overflow

Any invocation of `transfer(recipient, amount)` must fail if it causes the balance of the `recipient` address to overflow.

Specification:

```
requires recipient != msg.sender;
requires balanceOf(recipient) + amount > type(uint256).max;
ensures !\result;
```

erc20-transfer-revert-zero

Any call of the form `transfer(recipient, amount)` must fail if the recipient address is the zero address.

Specification:

```
ensures \old(recipient) == address(0) ==> !\result;
```

Properties related to function `allowance`

erc20-allowance-change-state

Function `allowance` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc20-allowance-correct-value

Invocations of `allowance(owner, spender)` must return the allowance that address `spender` has over tokens held by address `owner`.

Specification:

```
ensures \result == allowance(\old(owner), \old(spender));
```

erc20-allowance-succeed-always

Function `allowance` must always succeed, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `balanceOf`

erc20-balanceof-change-state

Function `balanceOf` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc20-balanceof-correct-value

Invocations of `balanceOf(owner)` must return the value that is held in the contract's balance mapping for address `owner`.

Specification:

```
ensures \result == balanceOf(\old(account));
```

erc20-balanceof-succeed-always

Function `balanceOf` must always succeed if it does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `totalSupply`

erc20-totalsupply-change-state

The `totalSupply` function in contract StakedGGBR must not change any state variables.

Specification:

```
assignable \nothing;
```

erc20-totalsupply-correct-value

The `totalSupply` function must return the value that is held in the corresponding state variable of contract StakedGGBR.

Specification:

```
ensures \result == totalSupply();
```

erc20-totalsupply-succeed-always

The function `totalSupply` must always succeeds, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `approve`

erc20-approve-correct-amount

All non-reverting calls of the form `approve(spender, amount)` that return `true` must correctly update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount`.

Specification:

```
requires spender != address(0);
ensures \result ==> allowance(msg.sender, \old(spender)) == \old(amount);
```

erc20-approve-false

If function `approve` returns `false` to signal a failure, it must undo all state changes that it incurred before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-approve-never-return-false

The function `approve` must never returns `false`.

Specification:

```
ensures \result;
```

erc20-approve-revert-zero

All calls of the form `approve(spender, amount)` must fail if the address in `spender` is the zero address.

Specification:

```
ensures \old(spender) == address(0) ==> !\result;
```

erc20-approve-succeed-normal

All calls of the form `approve(spender, amount)` must succeed, if

- the address in `spender` is not the zero address and
- the execution does not run out of gas.

Specification:

```
requires spender != address(0);
ensures \result;
reverts_only_when false;
```

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevate Your Web3 Journey

CertiK is the largest Web3 security platform combining formal verification with audits and comprehensive security solutions.