Goldcoin Audit Report

Zach Katz

May 4, 2025

Goldcoin Protocol Audit Report

Goldcoin Audit Report

Zach Katz

May 4, 2025

Prepared by: Zach Katz

Table of Contents

Protocol Summary

Goldcoin is a completely decentralized, **hypothetical**, gold-backed stablecoin exchange protocol created by Zach Katz. Goldcoin is pegged to the price of one ounce of gold 1:1 and features a liquidity pool where liquidity providers receive 50 basis points of their total relative staked value for each Goldcoin minted. Liquidity is staked in ETH, not Goldcoin, and thanks to this, holders of Goldcoin can exchange their holdings back for ETH.

Disclaimer

Zach makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by him is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

Scope

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	1
Medium	1
Low	1
Info	1
Total	4

Findings

High

[H-1] Reentrancy in `GCLiquidityPool::_distributeRewards`

Description: `provider.pendingRewards = 0` is set AFTER making a payable call out of `GCLiquidityPool` to liquidity providers, leaving the contract open to

reentrancy. Even though `GCLiquidityPool::_distributeRewards` is an internal function and only called every 30 days via automation, a hacker can reenter via `GCLiquidityPool::claimRewards` to claim double the amount of rewards.

Impact: Attackers exploiting this vulnerability can claim double the amount of rewards for themselves.

Proof of Concept:

```
contract Attacker {
    GCLiquidityPool public liquidityPool;

    // triggered when rewards are automatically distributed
    fallback() {
        liquidityPool.claimRewards();
    }

    stakeLiquidity(GCLiquidityPool _liquidityPool) external payable {
        liquidityPool = _liquidityPool;
        liquidityPool.stake{value: 1 ether};
    }
}
```

Recommended Mitigation: Following the checks-effects-interactions pattern will nullify this vulnerability. In practice this looks like:

```
function _distributeRewards() internal {
        // [AUDIT] Cached array length -- in several places
        for (uint256 i = 0; i < s_providerList.length; i++) {
            address providerAddress = s_providerList[i];
            _updateRewards(providerAddress);
            ProviderInfo storage provider = s_providers[providerAddress];
            uint256 reward = provider.pendingRewards;
            if (reward > 0) {
+               provider.pendingRewards = 0;
                (bool success, ) = payable(providerAddress).call{value: reward}(
                    ""
                );
                require(success, "Transfer failed");
-               provider.pendingRewards = 0;
```

Medium

[M-1] DoS vulnerability where calls in a loop are contigent on transfer success in `GCLiquidityPool::_distributeRewards`

Description: Payable calls made by looping through all liquidity providers can be a DoS attack vector if a provider has `revert()` in their fallback function.

Impact: Can prevent automatic disbursement of staking rewards from ever happening for all if there is one adversarial actor.

Proof of Concept:

```
contract Attacker {
    GCLiquidityPool public liquidityPool;

    // triggered when rewards are automatically distributed
    fallback() {
        revert();
    }

    stakeLiquidity(GCLiquidityPool _liquidityPool) external payable {
        liquidityPool = _liquidityPool;
        liquidityPool.stake{value: 1 ether};
    }
}
```

Recommended Mitigation: Remove the `require` statement that allows the entire distributiont to be reverted if the call is unsuccessful

```
if (reward > 0) {
    (bool success, ) = payable(providerAddress).call{value: reward}(
        ""
    );
-   require(success, "Transfer failed");
    provider.pendingRewards = 0;
}
```

Low

[L-1] Reentrancy in `Goldcoin::exchangeAndBurn`

Description: `Goldcoin::_burn` is called after paying out the exchange of goldcoin tokens that happens in `GCLiquidityProvider::handleBurn`, leaving the contract open to reentrancy.

Impact: Currently there are no real consequences for this vulnerability, as the most an attacker could do would be to revert the burn in their fallback, but if changes are made in the future to the way burning of tokens is carried out, the impact could be more serious.

Recommended Mitigation: Following the checks-effects-interactions pattern will nullify this vulnerability. In practice this looks like:

```
function exchangeAndBurn(uint256 amount) external {
        require(amount > 0, "Amount must be greater than 0");
        require(balanceOf(msg.sender) >= amount, "Insufficient balance");
```

```
+       _burn(msg.sender, amount);
        s_liquidityPool.handleBurn(getGoldcoinInWei(amount), msg.sender);
-       _burn(msg.sender, amount);
    }
```

Informational

[I-1] Rewards are not paid out when provider completely exits the pool in `GCLiquidityPool::withdraw`

Description: When a provider completely exits their position by unstaking the full amount, their rewards are not distributed.

Impact: Providers have to manually claim their pending rewards, automatic distribution will not work as they are removed from `s_providerList`.

Proof of Concept:

Recommended Mitigation: Distribute rewards at the same time a liquidity provider exits their position completely.