Security Review Report

Goat.fi (DeFi Yield Optimizer)

Led by Tigran Piliposyan at 14.02.2024

About Tigran Piliposyan

Tigran Piliposyan (tpiliposian) is a security researcher specializing in smart contract audits and security consulting.

Currently, he holds the role of Security Researcher at Hexens and a Core Contributor at Security Wiki.

Previously, he held roles in the financial sphere for over a decade: LinkedIn.

Feel free to connect with him at:

- X/Twitter - tpiliposian
- Telegram - tpiliposian

Disclaimer

A smart contract security review is a comprehensive attempt to uncover vulnerabilities, but it cannot establish their complete absence. The process is bounded by time, resources, and expertise, aiming to identify as many potential issues as possible. However, it does not assure absolute security for the protocol.

Severity classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- High - Funds are directly at risk or a severe disruption of the protocol's core functionality.
- Medium - Funds are indirectly at risk or some disruption of the protocol's functionality.
- Low - Funds are not at risk.

Likelihood

- High - Highly likely to occur.
- Medium - Might occur under specific conditions.
- Low - Unlikely to occur.

About Goat Protocol

The Goat Protocol is a decentralized yield optimizer. It allows users, DAOs, and other protocols to earn the yield on their digital assets by auto-compounding the rewards into more of what they've deposited.

Summary & Scope

The Goat.fi repository was audited at commit 724d610f4f5d7bb9abf0b965a66cbf0ec809953b.

The following contracts were in scope:

1. `src/infra/GoatRewardPool.sol` (~200 nSLOC)

After the completion of the fixes, the following commits were reviewed:

- https://github.com/goatfi/contracts/pull/3/commits/c37679b28b7fe4f998db4b7b431c71f63e18ee58
- https://github.com/goatfi/contracts/pull/3/commits/601ae2d9d1eff034da3806f1faf5a841c5c94cfc
- https://github.com/goatfi/contracts/pull/3/commits/4ea4bb31dd6c7a09455a72fae7b0222af6dc957f

Summary of Findings

Severity	Issues Found
High	0
Medium	2
Low	0
Informational	1

Summary

Identifier	Title	Severity	Status
[M-01]	Stake with permit can be blocked	Medium	Fixed
[M-02]	Inconsistent decimal place handling in reward calculations	Medium	Fixed
[I-01]	Unnecessary initialization of constant in the constructor	Informational	Fixed

Findings

[M-01] Stake with permit can be blocked

Description

Path: `GoatRewardPool.sol:stakeWithPermit()`
https://github.com/goatfi/contracts/blob/724d610f4f5d7bb9abf0b965a66cbf0ec809953b/src/infra/GoatRewardPool.sol#L124-L136

The `stakeWithPermit` function internally calls `permit()` function from `IERC20Permit(address(stakedToken))`. However, this flow exposes `stakeWithPermit` to a griefing attack, where an attacker can forcibly block the victim's transaction.

```
    function stakeWithPermit(
        address _user,
        uint256 _amount,
        uint256 _deadline,
        uint8 _v,
        bytes32 _r,
        bytes32 _s
    ) external update(_user) {
        IERC20Permit(address(stakedToken)).permit(
            _user, address(this), _amount, _deadline, _v, _r, _s
        );
        _stake(_user, _amount);
    }
```

Attack scenario: the attacker front-runs the victim's transaction, extracts parameters from the mempool, and places a transaction that directly calls `IERC20Permit(address(stakedToken).permit()` with parameters given by the victim. Consequently, the victim's transaction reverts since parameters have already been used for `permit()` in the attacker's transaction.

Recommendation

You can check how `The Graph` solved this issue by adding their own `_permit` like this: https://github.com/edgeandnode/billing-contracts/blob/4aa2c70702da61c67b9d58cf066773a3b1dde539/contracts/BillingConnector.sol#L268-L299

Consider adding `try/cath`, or `if` block, so if the `permit()` is already called, just check the allowance of `msg.sender` and skip the call to `pemit()`:

An example from `The Grapgh`:

```
        IERC20WithPermit token = IERC20WithPermit(address(graphToken));
        // Try permit() before allowance check to advance nonce if possible
        try token.permit(_owner, _spender, _value, _deadline, _v, _r, _s) {
            return;
        } catch Error(string memory reason) {
            // Check for existing allowance before reverting
            if (token.allowance(_owner, _spender) >= _value) {
                return;
            }

            revert(reason);
        }
```

[M-02] Inconsistent decimal place handling in reward calculations

Description

https://github.com/goatfi/contracts/blob/724d610f4f5d7bb9abf0b965a66cbf0ec809953b/src/infra/GoatRewardPool.sol#L396-L403

The comment and calculations in the `_earned` function of the `GoatRewardPool.sol` contract appear to be inconsistent with each other regarding the decimal precision of the reward tokens. While the comment states that `rewardPerTokenStored` is in 18 decimals, the calculations use a fixed divisor of `1e30`, which suggests a different decimal precision. This inconsistency may lead to incorrect reward calculations.

```
    /// @param rewardPerTokenStored Stored reward value per staked token in 18 decimals
    /// @param userRewardPerTokenPaid Stored reward value per staked token in 18 decimals at the
    /// last time a user was paid the reward
```

```
    function _rewardPerToken(address _reward) private view returns (uint256 rewardPerToken) {
        RewardInfo storage rewardData = _getRewardInfo(_reward);
        if (totalSupply() == 0) {
            rewardPerToken = rewardData.rewardPerTokenStored;
        } else {
            rewardPerToken = rewardData.rewardPerTokenStored + Math.mulDiv(
                (_lastTimeRewardApplicable(rewardData.periodFinish) - rewardData.lastUpdateTime),
                rewardData.rate * 1e30,
                totalSupply()
            );
        }
    }

    /// @dev Calculate the reward amount earned by the user
    /// @param _user Address of the user
    /// @param _reward Address of the reward
    /// @return earnedAmount Amount of reward earned by the user
    function _earned(address _user, address _reward) private view returns (uint256 earnedAmount) {
        RewardInfo storage rewardData = _getRewardInfo(_reward);
        earnedAmount = rewardData.earned[_user] + Math.mulDiv(
            balanceOf(_user),
            (_rewardPerToken(_reward) - rewardData.userRewardPerTokenPaid[_user]),
            1e30
        );
    }
```

Recommendation

Ensure that comments accurately reflect the decimal precision of the reward tokens.

[I-01] Unnecessary initialization of constant in the constructor

Description

https://github.com/goatfi/contracts/blob/724d610f4f5d7bb9abf0b965a66cbf0ec809953b/src/infra/GoatRewardPool.sol#L106

In the constructor of the `GoatRewardPool.sol` contract, the variable `rewardMax` is set to a value of 10. However, this value is not changed anywhere else in the

contract. Initializing a constant value in the constructor is unnecessary and can be optimized.

```
    constructor(address _stakedToken) ERC20("Staked GOA", "stGOA") Ownable(msg.sender) {
        stakedToken = IERC20(_stakedToken);
        rewardMax = 10;
    }
```

Recommendation

Consider declaring `rewardMax` as an internal constant directly in the contract, rather than initializing it in the constructor.