verichains

SECURITY AUDIT OF

POCO STAKING SMART CONTRACTS

PUBLIC REPORT

September 16, 2021

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ACRONYMS AND ABBREVIATIONS

NAME	DESCRIPTION
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
EVM	Ethereum Virtual Machine
LP	Liquidity Provider

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Sep 16, 2021. We would like to thank the Poco team for trusting Verichains Lab in audit smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Poco Staking smart contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

The assessment did not identify any vulnerability issue in Poco Staking smart contract code.

Overall, the code reviewed is of good quality, written with the awareness of smart contract development best practices.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Poco and Poco Staking

Poco brings you into the new gaming world. Let's immerse yourself in Pocoland when leading the powerful team with 5 Poco warrants owning different elements, defeat your enemies then collect the huge reward by POCO token. Poco Staking is the official staking platform of Poco.

More information at https://pocoland.com/staking.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the core smart contracts of Poco Staking. It was conducted on the source code we're provided by the Poco team.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in Table 1, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.
INFO	A recommendation that should be fixed but it's not mandatory.

Table 1: Vulnerability severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Summary

There are four interfaces, one library and four contracts in the Poco Staking source codes:

- `IERC20 (interfaces/IERC20.sol)`: this is the interface of ERC-20 standard as defined in EIP-20.
- `IERC20Detailed (interfaces/IERC20Detailed.sol)`: this is the interface for ERC20 including metadata.
- `IStakedPoco (interfaces/IStakedPoco.sol)`: this is the interface to interact with the StakedPoco contract.
- `IDistributionManager (interfaces/IDistributionManager.sol)`: this is the interface to interact with the PocoDistributionManager contract.
- `DistributionTypes (libs/DistributionTypes.sol)`: this is a library which defines distribution types to be used in the `PocoDistributionManager` contract.
- `ERC20WithSnapshot (libs/ERC20WithSnapshot.sol)`: this is a custom implementation of ERC20 including snapshots of balances on transfer-related actions.
- `PocoDistributionManager (stake/PocoDistributionManager.sol)`: this is the smart contract to manage multiple staking distributions.
- `StakedToken (staking/StakedToken.sol)`: this is the smart contract to stake Poco token, tokenize the position and get rewards, inheriting from a distribution manager contract.
- `StakedPoco (staking/StakedPoco.sol)`: this is the contract which defined the Poco's staked token by extending the `StakedToken` contract.
- `VersionedInitializable (utils/VersionedInitializable.sol)`: this is a helper contract to support initializer functions which be used in the `StakedToken` contract.

2.2. Findings

During the audit process, the audit team did not identify any security vulnerability issue in the Poco Staking Smart Contracts.

2.3. Additional notes and recommendations

2.3.1. StakedToken – Improve zero amount checking in redeem function

The redeemed amount will be zero if the input amount is zero or the current balance of this user is zero. If the user's balance is zero, we can skip the rest of the logic for gas saving.

```
function redeem(address to, uint256 amount) external override {
    require(amount != 0, 'INVALID_ZERO_AMOUNT');
    uint256 balanceOfMessageSender = balanceOf(msg.sender);
    uint256 amountToRedeem = (amount > balanceOfMessageSender) ? balanceOfMessageSender
: amount;
    _updateCurrentUnclaimedRewards(msg.sender, balanceOfMessageSender, true);
    _burn(msg.sender, amountToRedeem);
    IERC20(STAKED_TOKEN).safeTransfer(to, amountToRedeem);
    emit Redeem(msg.sender, to, amountToRedeem);
}
```

RECOMMENDATION

We should check the `amountToRedeem` variable as below.

```
function redeem(address to, uint256 amount) external override {
    // require(amount != 0, 'INVALID_ZERO_AMOUNT');
    uint256 balanceOfMessageSender = balanceOf(msg.sender);
    uint256 amountToRedeem = (amount > balanceOfMessageSender) ? balanceOfMessageSender
: amount;
    require(amountToRedeem > 0, 'INVALID_ZERO_AMOUNT');
    _updateCurrentUnclaimedRewards(msg.sender, balanceOfMessageSender, true);
    _burn(msg.sender, amountToRedeem);
    IERC20(STAKED_TOKEN).safeTransfer(to, amountToRedeem);
    emit Redeem(msg.sender, to, amountToRedeem);
}
```

UPDATE

This finding has been acknowledged by Poco team.

2.3.2. ERC20WithSnapshot – Unnecessary integer overflow checking

In `_writeSnapshot` function, the `ownerCountOfSnapshots` variable has been checked with `ownerCountOfSnapshots != 0`. So, the `ownerCountOfSnapshots.sub(1)` here is unnecessary. Also, the range of `ownerCountOfSnapshots` variable is large enough so that we can skip using the **`SafeMath`** for gas saving.

```
function _writeSnapshot(
    address owner,
    uint128 oldValue,
    uint128 newValue
) internal virtual {
    uint128 currentBlock = uint128(block.number);

    uint256 ownerCountOfSnapshots = _countsSnapshots[owner];
    mapping(uint256 => Snapshot) storage snapshotsOwner = _snapshots[owner];

    // Doing multiple operations in the same block
    if (
        ownerCountOfSnapshots != 0 &&
        snapshotsOwner[ownerCountOfSnapshots.sub(1)].blockNumber == currentBlock
    ) {
        snapshotsOwner[ownerCountOfSnapshots.sub(1)].value = newValue;
    } else {
        snapshotsOwner[ownerCountOfSnapshots] = Snapshot(currentBlock, newValue);
        _countsSnapshots[owner] = ownerCountOfSnapshots.add(1);
    }

    emit SnapshotDone(owner, oldValue, newValue);
}
```

RECOMMENDATION

We should update the code as below.

```
function _writeSnapshot(
    address owner,
    uint128 oldValue,
    uint128 newValue
) internal virtual {
    uint128 currentBlock = uint128(block.number);

    uint256 ownerCountOfSnapshots = _countsSnapshots[owner];
    mapping(uint256 => Snapshot) storage snapshotsOwner = _snapshots[owner];

    // Doing multiple operations in the same block
    if (
        ownerCountOfSnapshots != 0 &&
        snapshotsOwner[ownerCountOfSnapshots - 1].blockNumber == currentBlock
    ) {
        snapshotsOwner[ownerCountOfSnapshots - 1].value = newValue;
    } else {
        snapshotsOwner[ownerCountOfSnapshots] = Snapshot(currentBlock, newValue);
        _countsSnapshots[owner] = ownerCountOfSnapshots + 1;
    }

    emit SnapshotDone(owner, oldValue, newValue);
}
```

UPDATE

This finding has been acknowledged by Poco team.

3. VERSION HISTORY

VERSION	DATE	STATUS/CHANGES	CREATED BY
1.0	Sep 16, 2021	Private report	Verichains Lab
1.1	Sep 16, 2021	Public report	Verichains Lab

APPENDIX A: OVERVIEW FUNCTION CALL GRAPH

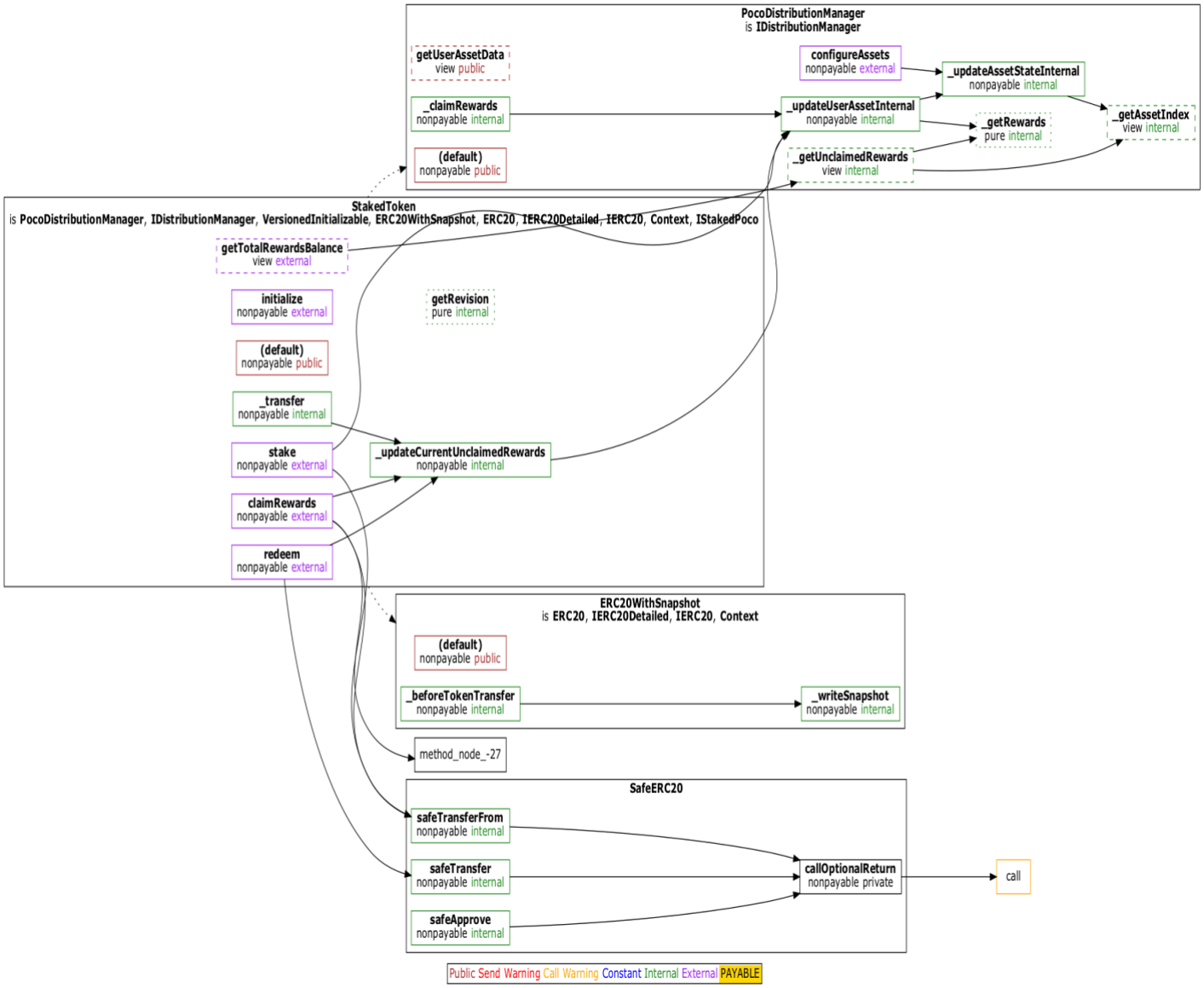

Figure 1: function call graph of StakedToken contract