ZEALYNX SECURITY

Web3 Security & Smart Contract Development

BadgerDAO

Security Assessment

October 11th, 2024 - Prepared by Zealynx Security

ZealynxSecurity@protonmail.com

Sergio	@Seecoalba
Bloqarl	@TheBlockChainer
Alejandro	@mevquant

Contents

1. About Zealynx

Zealynx Security, founded in January 2024 by Bloqarl and Sergio, specializes in smart contract audits, development, and security testing using Solidity and Rust. Our services include comprehensive smart contract audits, smart contract fuzzing, and formal verification test suites to enhance security reviews and ensure robust contract functionality. We are trusted by clients such as Glif, Shieldify, AuditOne, Bastion Wallet, Side.xyz, Possum Labs, and Aurora (NEAR Protocol-based).

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our blog, Bloqarl's blog, and Sergio's blog.

Zealynx has achieved public recognition, including a Top 5 position in the Beanstalk Audit public contest, rewarded with $8k. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

A time-boxed independent security assessment of the BadgerDAO was done by Zealynx Security, with a focus on the security aspects of the application's implementation.
We performed the security assessment based on the agreed scope during 1 week, following our approach and methodology. Based on the scope and our performed activities, our security assessment revealed 1 **Medium**, 3 **Low** and 2 **Informational** security issues.

3.2 About BadgerDAO

Website : BadgerDAO
Docs: Badger docs

Badger is a decentralized autonomous organization (DAO) with a single purpose: build the products and infrastructure necessary to accelerate Bitcoin as collateral across other blockchains.

It's meant to be an ecosystem DAO where projects and people from across DeFi can come together to collaborate and build the products our space needs. Shared ownership in the DAO will allow builders to have aligned incentives while decentralized governance can ensure those incentives remain fair to all parties. The idea is less competing and more collaborating.

3.3 Audit Scope

The code under review is composed of multiple smart contracts written in Solidity language and includes 701 nSLOC- normalized source lines of code. The core contracts on the scope are the following:

4. Audit Methodology:

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough **Fuzz** and **Invariant Testing, Formal Verification,** and meticulous **manual security review**.

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality:** We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. **Best Practices:** Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. **Documentation and Comments:** We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

Over the course of 1 week, BadgerDAO team engaged with Zealynx Security to review their smart contracts. In this period of time, a total of **6 issues** were found.

Summary of Findings

Vulnerability	Severity
[M-1] Precision loss in _cycleEnd calculation	Medium
[L-01] Emit Events for Transparency in Main Function	Low
[L-02] Enhance Permit Failure Handling in depositWithSignature Function	Low
[L-03] Blacklist as Alternative to Minting Fee	Low
[I-01] Use Custom Errors Instead of Require Statements for Gas Optimization and Clarity	Informational
[I-02] Use camelCase for Immutable Variables	Informational

7. Findings

7.1. Medium Severity Findings

[M-1] Precision loss in _cycleEnd calculation

Description

In the previewSyncRewards() function, there is a potential precision loss when calculating the value of _cycleEnd. The calculation uses division and multiplication operations with integers, which can result in a loss of precision due to rounding.

Impact

The precision loss in the calculation of _cycleEnd could result in reward cycles that do not perfectly align with the expected time intervals. This leads to small discrepancies in the distribution of rewards over time, as the amount of rewards will be smaller than it is meant to be.

Proof Of Concept

```
function test_PrecisionLossIn_cycleEnd(uint256 _amount, uint256 _timePassed) public {
        vm.assume(_amount > 0 && _amount <= 1000 ether);
        vm.assume(_timePassed > 1 days && _timePassed <= 7 days);

        uint40 cycleEndBefore = stakedEbtc.__rewardsCycleData().cycleEnd;
        uint40 lastSyncBefore = stakedEbtc.__rewardsCycleData().lastSync;
        uint216 rewardCycleAmountBefore = stakedEbtc.__rewardsCycleData().rewardCycleAmount;

        vm.prank(bob);
        uint256 shares = stakedEbtc.deposit(_amount, bob);

        uint256 cycleLength = stakedEbtc.REWARDS_CYCLE_LENGTH();
        uint256 timeBefore = block.timestamp + _timePassed;

        // Warp to a time near the end of the cycle
        uint256 timeNearCycleEnd = timeBefore + cycleLength - (cycleLength / 40);
        vm.warp(timeNearCycleEnd);

        uint256 _timestamp = block.timestamp;
        uint40 cycleEndUnoptimized = (((_timestamp + cycleLength) / cycleLength) * cycleLength).safeCastTo40();
        uint40 cycleEndOptimized = (((_timestamp + cycleLength))).safeCastTo40();

        if (cycleEndOptimized - _timestamp < cycleLength / 40) {
            cycleEndOptimized += cycleLength.safeCastTo40();
        }

        if (cycleEndUnoptimized - _timestamp < cycleLength / 40) {
            cycleEndUnoptimized += cycleLength.safeCastTo40();
        }

        stakedEbtc.syncRewardsAndDistribution();

        uint40 cycleEndInContract = stakedEbtc.__rewardsCycleData().cycleEnd;
        uint40 lastSync = stakedEbtc.__rewardsCycleData().lastSync;
        uint216 rewardCycleAmount = stakedEbtc.__rewardsCycleData().rewardCycleAmount;


        console.log("=== Cycle End Comparison ===");
        console.log("Contract Cycle End:", cycleEndInContract);
        console.log("Test Optimized Cycle End:", cycleEndOptimized);
        console.log("Test Unoptimized Cycle End:", cycleEndUnoptimized);
        console.log("cycleEndUnoptimized != cycleEndOptimized");
        console.log("========================");

        uint256 rest ;
        if (cycleEndOptimized > cycleEndUnoptimized) {
            rest = cycleEndOptimized - cycleEndUnoptimized;
        } else
        rest = cycleEndUnoptimized - cycleEndOptimized;

        console.log("=== Precision Analysis ===");
        console.log("Difference between Optimized and Unoptimized:", rest);
        console.log("========================");

        assertEq(cycleEndInContract, cycleEndOptimized, "Cycle end should match the calculated value");
        assertEq(cycleEndUnoptimized,cycleEndOptimized, "Precision loss");
    }
```

Console output:

```
[FAIL: Cycle end should match the calculated value: 1209600 != 1410866;
  === Cycle End Comparison ===
  Contract Cycle End: 1209600
  Test Optimized Cycle End: 1410866
  Test Unoptimized Cycle End: 1209600
  cycleEndUnoptimized != cycleEndOptimized
  ========================
  === Precision Analysis ===
  Difference between Optimized and Unoptimized: 201266
  ========================
```

Mitigation

Simplify the formula for calculating _cycleEnd by removing the redundant operation of multiplying and dividing for the same variable REWARDS_CYCLE_LENGTH.

The optimized formula would be:

```
uint40 _cycleEnd = (_timestamp + REWARDS_CYCLE_LENGTH).safeCastTo40();
```

7.2. Low Severity Findings

[L-01] Emit Events for Transparency in Main Function

Description

The sweep function, which handles unauthorized donations and extra tokens, doesn't emit an event. Consider adding an event emission for better transparency and easier off-chain tracking.

Impact

- Decreased transparency for users and auditors
- Hindered historical auditing of contract actions
- Potential difficulties in integrating with external systems

Mitigation

Add an event emission at the end of the sweep function. For example:

```
event Swept(address indexed token, uint256 amount);

function sweep(address token) external requiresAuth {
// ... existing function body ...

    emit Swept(token, sweptAmount);
}
```

[L-02] Enhance Permit Failure Handling in depositWithSignature Function

Description

The depositWithSignature function currently uses a try-catch block to handle potential permit failures, which is good for mitigating front-running attacks. However, the catch block is empty, which might lead to silent failures and unexpected behavior.

Impact

In cases where the permit fails and the user doesn't have sufficient allowance, the transaction might fail at a later stage (during the actual deposit), wasting gas and causing confusion.

Mitigation

Implement an allowance check in the catch block. This ensures that if the permit fails (possibly due to front-running), the function can still proceed if the user has sufficient allowance. If not, it will revert with a clear error message.

```
try
    asset.permit({
        owner: msg.sender,
        spender: address(this),
        value: _amount,
        deadline: _deadline,
        v: _v,
        r: _r,
        s: _s
    })
{ } catch {
// Permit failed (possibly due to front-running), check allowance
    uint256 allowance = IERC20(asset).allowance(msg.sender, address(this));
    if (allowance < _assets) {
        revert InsufficientAllowance(_assets, allowance);
    }
// If allowance is sufficient, continue with deposit despite permit failure
}
```

This modification maintains protection against front-running attacks while also ensuring that transactions can proceed if sufficient allowance exists, improving user experience and providing clearer error messages when both permit and allowance are insufficient.

[L-3] Blacklist as Alternative to Minting Fee

Description

The eBTC protocol is considering implementing a minting fee to prevent users from gaming the system by minting eBTC right after a rebase and repaying it just before the next rebase, thereby bypassing the Protocol Yield Splitting (PYS). However, this approach impacts all users, including those not attempting to game the system. An alternative approach using a blacklist mechanism could be considered as a first step or complete alternative.

Impact

Implementing a minting fee affects all users, including long-term holders and legitimate short-term users who aren't trying to game the system.
It also may discourage legitimate short-term usage of the protocol, potentially reducing overall liquidity and utility.
Worth noting that adding a minting fee, while it generates additional revenue for the protocol, it does so at the cost of user-friendliness.

Mitigation

Implementing a blacklist approach as an alternative or precursor is recommended as only affects users identified as attempting to game the system, preserving the experience for legitimate users

7.3. Informational Findings

[I-01] Use Custom Errors Instead of Require Statements for Gas Optimization and Clarity

Description

In multiple places in the contract, require statements are used for validation. Consider replacing these with custom errors for gas optimization and improved clarity in error messages.

Affected locations:

- Constructor:
  require(_feeRecipient != address(0))
- Deposit function:
  require((_shares = super.previewDeposit(assetsNoFee)) != 0, "ZERO_SHARES")

Suggestion

Replace require statements with custom errors. For example:

```
error ZeroAddress();
error ZeroShares();

// In constructor
if (_feeRecipient == address(0)) revert ZeroAddress();

// In deposit function
if ((_shares = super.previewDeposit(assetsNoFee)) == 0) revert ZeroShares();
```

This change will save gas and provide more informative error messages.

[I-02] Use camelCase for Immutable Variables

Description

The contract uses uppercase for the immutable variable FEE_RECIPIENT. While this is commonly used for constants, it's generally recommended to use camelCase for immutable variables to distinguish them from constants.

Suggestion

Rename FEE_RECIPIENT to feeRecipient for consistency with Solidity naming conventions.