// Security Assessment

09.22.2025 - 09.23.2025

ERC20 Diff

Bonzo Finance

HALBORN

ERC20 Diff - Bonzo Finance

Prepared by: HALBORN

Last Updated 10/01/2025

Date of Engagement: September 22nd, 2025 - September 23rd, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	0	0	2	1

TABLE OF CONTENTS

1. INTRODUCTION

`Bonzo Finance` engaged `Halborn` to conduct a security assessment on their smart contracts beginning on September 22nd, 2025 and ending on September 23rd, 2025. The security assessment was scoped to smart contracts in the GitHub repository provided to the `Halborn` team. Commit hashes and further details can be found in the Scope section of this report.

2. ASSESSMENT SUMMARY

The team at `Halborn` assigned a full-time security engineer to assess the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, `Halborn` identified some improvements to reduce the likelihood and impact of risks, which were completely addressed by the `Bonzo Finance team`:

- `Remove payable modifiers from functions if native tokens are not supported, or implement proper handling if they are.`
- `Introduce controlled recovery mechanisms for authorized rescue of stuck native and ERC20 tokens.`
- `Add zero-address validation to the recipient parameter during withdrawals to prevent accidental fund loss.`

3. CAVEATS

This assessment only covers the changes introduced between commit `5586803` and commit `bb5a582`. All other parts of the codebase are treated as black boxes, and it is assumed that any required security mechanisms are properly implemented elsewhere in the system. This includes, but is not limited to, input validation, business logic enforcement, and access control.

Differential audits focus on identifying security risks introduced or modified in a specific set of changes. While some components added in the diff may interact with each other or with existing logic, these interactions are only reviewed when they are explicitly visible within the diff itself. Implicit assumptions or cross-component dependencies may fall outside the scope unless clearly surfaced through changes or supporting context.

For a more complete understanding of the protocol's security posture and the behavior of interconnected components, a full-scope assessment is recommended.

4. TEST APPROACH AND METHODOLOGY

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Static Analysis of security for scoped contract, and imported functions. (Slither).
- Local or public testnet deployment (Hardhat, Remix IDE).

5. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

5.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

5.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9

SEVERITY	SCORE VALUE RANGE
Low	2 - 4.4
Informational	0 - 1.9

6. SCOPE

REPOSITORY

(a) Repository: bonzo-finance-contracts

(b) Assessed Commit ID: bb5a582

(c) Items in scope:

- contracts/misc/WHBARGateway.sol
- contracts/protocol/tokenization/AToken.sol
- contracts/protocol/lendingpool/LendingPool.sol

Out-of-Scope: Third party dependencies and economic attacks. Furthermore, this assessment was limited to the differential changes between commit 5586803 and commit bb5a582. Components outside of the reviewed diff, pre-existing logic not modified by the changes, system-wide interactions not directly surfaced in the diff context, and external assumptions regarding input validation, business rules, and access controls were explicitly out of scope.

REMEDIATION COMMIT ID:

- ca99583

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PAYABLE MODIFIER IN FUNCTIONS WITHOUT NATIVE TOKEN HANDLING LOGIC	LOW	SOLVED - 09/29/2025
LACK OF EMERGENCY RECOVERY MECHANISMS FOR NATIVE AND ERC20 TOKENS	LOW	SOLVED - 09/29/2025
POTENTIAL LOSS OF USER FUNDS DUE TO INSUFFICIENT INPUT VALIDATION	INFORMATIONAL	SOLVED - 09/29/2025

8. FINDINGS & TECH DETAILS

8.1 PAYABLE MODIFIER IN FUNCTIONS WITHOUT NATIVE TOKEN HANDLING LOGIC

// LOW

Description

The `deposit()` and `repay()` functions in the `LendingPool` contract have the `payable` modifier but do not handle native token transfers, causing any native tokens sent to these functions to become permanently locked in the contract. This creates a potential loss of funds for users who mistakenly send native tokens when calling these functions.

When users call these functions with `msg.value > 0`, the native tokens are transferred to the `LendingPool` contract but remain unprocessed. The functions proceed with their intended ERC20 token operations while the native tokens become permanently locked, as there is no mechanism to retrieve them.

Code Location

The `deposit` and `repay` functions marked as payable without native token handling logic:

```
111 function deposit(
112     address asset,
113     uint256 amount,
114     address onBehalfOf,
115     uint16 referralCode
116 ) external payable override whenNotPaused {
117 ...
118 }
```

```
246 function repay(
247     address asset,
248     uint256 amount,
249     uint256 rateMode,
250     address onBehalfOf
251 ) external payable override whenNotPaused returns (uint256) {
252 ...
253 }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (3.4)

Recommendation

Remove the `payable` modifier from both functions if native token support is not intended, or implement proper native token handling if such functionality is desired.

Remediation Comment

SOLVED: The **Bonzo Finance team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/Bonzo-Labs/bonzo-finance-contracts/commit/ca995838db5c78d0ae0301c7619249e48b7491c3

8.2 LACK OF EMERGENCY RECOVERY MECHANISMS FOR NATIVE AND ERC20 TOKENS

// LOW

Description

The `WHBARGateway` contract lacks emergency recovery functions for both native `HBAR` and `ERC20` tokens that may become stuck in the contract through force-feeding attacks (e.g., `selfdestruct`) or accidental direct transfers. Although the contract restricts normal native token deposits via the `receive()` function, it provides no mechanism to recover tokens that bypass these restrictions, leading to potential permanent fund lockup.

Tokens can become stuck in the contract under the following scenarios:

- **Native Token:** *Force-Feeding via* `Selfdestruct` — Any contract can call `selfdestruct` on itself and set the `WHBARGateway` as the recipient, bypassing receive() restrictions and forcing `HBAR` into the contract.
- **Non-Native Token Issues:** *Accidental ERC20 Transfers* — Users may mistakenly transfer ERC20 tokens directly to the gateway (e.g., `token.transfer(gateway, amount)`), resulting in tokens being held by the contract.

In both cases, the absence of owner-controlled emergency recovery functions causes permanent and unrecoverable fund lockup.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (3.4)

Recommendation

Introduce controlled recovery mechanisms that allow an authorized entity (contract owner) to rescue both native and ERC20 tokens stuck in the contract. Consider implementing:

- A `recoverERC20(address token, uint256 amount, address recipient)` function restricted to an admin role.
- And also a `recoverNative(address recipient)` function to withdraw force-fed `HBAR`.

Remediation Comment

SOLVED: The **Bonzo Finance team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/Bonzo-Labs/bonzo-finance-contracts/commit/ca995838db5c78d0ae0301c7619249e48b7491c3

8.3 POTENTIAL LOSS OF USER FUNDS DUE TO INSUFFICIENT INPUT VALIDATION

// INFORMATIONAL

Description

The `withdrawHBAR` function in the `WHBARGateway.sol` contract lacks validation on the `to` parameter, allowing users to accidentally withdraw funds to the zero address, resulting in permanent loss of funds. The function accepts any address as the `to` parameter without performing basic validation checks. This creates a potential issue where users may accidentally pass address(0) due to:

- Frontend bugs or UI errors
- Copy-paste mistakes
- Uninitialized variables in calling contracts
- Malicious dApp interfaces

If a user mistakenly calls `withdrawHBAR` with the zero address as the recipient, the function still executes. The user's `aWHBAR` tokens are burned (via `transferFrom` and `withdraw`), `WHBAR` is unwrapped into native `HBAR`, and the `HBAR` is sent to the zero address. Since no recovery mechanism exists, the funds are permanently lost.

Code Location

Absence of zero address check on the `to` parameter in the `withdrawHBAR` function

```
38  function withdrawHBAR(address lendingPool, uint256 amount, address to) external {
39      IAToken aWHBAR = IAToken(ILendingPool(lendingPool).getReserveData(address(WHBAR)).aTokenAddress);
40      uint256 userBalance = aWHBAR.balanceOf(msg.sender);
41      uint256 amountToWithdraw = amount;
42      if (amount == type(uint256).max) {
43          amountToWithdraw = userBalance;
44      }
45      aWHBAR.transferFrom(msg.sender, address(this),    amountToWithdraw);
46      ILendingPool(lendingPool).withdraw(address(WHBAR),   amountToWithdraw, address(this));
47      WHBAR.withdraw(amountToWithdraw);
48     _safeTransferHBAR(to, amountToWithdraw);
49  }
```

```
105 function _safeTransferHBAR(address to, uint256 value) internal {
106     (bool success, ) = to.call{value: value}(new bytes(0));
107
108
```

```
        require(success, 'HBAR_TRANSFER_FAILED');
    }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (1.7)

Recommendation

Add an input validation in `withdrawHBAR` to revert on zero-address recipients, preventing fund loss without affecting existing functionality.

Remediation Comment

SOLVED: The **Bonzo Finance team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/Bonzo-Labs/bonzo-finance-contracts/commit/ca995838db5c78d0ae0301c7619249e48b7491c3
