// Security Assessment

03.10.2025 - 03.12.2025

Staking Module

Bonzo Finance

HALBORN

Staking Module - Bonzo Finance

Prepared by: HALBORN

Last Updated 03/13/2025

Date of Engagement by: March 10th, 2025 - March 12th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	1	1	3

TABLE OF CONTENTS

1. Introduction

`Bonzo team` engaged `Halborn` to conduct a security assessment on their smart contracts revisions started on March 10th, 2025 and ending on March 12th, 2024. The security assessment was scoped to the smart contracts 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 was provided 3 days for the engagement and assigned a security engineer to evaluate the security of the smart contract.

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 should be addressed by the `Bonzo team`. The main ones were the following:

- `Restrict the createXBonzoToken function to authorized accounts, to prevent unauthorized or malicious token creation.`
- `Ensure that the contract always holds enough Bonzo tokens to cover both staked amounts and accrued rewards.`

3. Test Approach And Methodology

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

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph,draw.io`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)

4. 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.

4.1 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:

$$E = \prod m_e$$

4.2 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 (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I: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:

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

4.3 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

5. SCOPE

FILES AND REPOSITORY

(a) Repository: bonzo-staking-module

(b) Assessed Commit ID: ec319c7

(c) Items in scope:

- BonzoStaking.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 5b54b86

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	1	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - UNAUTHORIZED CONTROL OF TOKEN RENEWAL VIA UNRESTRICTED AUTORENEWACCOUNT IN XBONZO CREATION	MEDIUM	SOLVED - 03/12/2025
HAL-02 - INSUFFICIENT BONZO FUNDING FOR ACCRUED REWARDS MIGHT CAUSE UNSTAKE FAILURES	LOW	ACKNOWLEDGED
HAL-03 - UNLOCKED PRAGMA COMPILER	INFORMATIONAL	SOLVED - 03/12/2025
HAL-04 - INCORRECT ORDER OF MODIFIERS: NONREENTRANT SHOULD PRECEDE ALL OTHER MODIFIERS	INFORMATIONAL	SOLVED - 03/12/2025
HAL-05 - DUPLICATE EXCHANGE RATE FUNCTIONS LEAD TO REDUNDANT CODE	INFORMATIONAL	SOLVED - 03/12/2025

7. FINDINGS & TECH DETAILS

7.1 (HAL-01) UNAUTHORIZED CONTROL OF TOKEN RENEWAL VIA UNRESTRICTED AUTORENEWACCOUNT IN XBONZO CREATION

// MEDIUM

Description

The `createXBonzoToken` function in the **BonzoStaking** contract is publicly callable and lacks access control, allowing any external account to invoke it. This function is intended to create the xBonzo staking receipt token using Hedera's HTS precompile. However, without proper restrictions, anyone can create xBonzo on behalf of the staking contract before the legitimate deployer does, but front-running the regarded transaction.

```
function createXBonzoToken(address _autoRenewAccount) external payable returns (address) {
  require(xBonzoToken == address(0), 'xBONZO token already created');

  IHederaTokenService.HederaToken memory tokenData;
  tokenData.name = 'xBONZO';
  tokenData.symbol = 'xBONZO';
  tokenData.treasury = address(this);
  tokenData.memo = 'xBONZO staking token';
  tokenData.tokenSupplyType = false;
  tokenData.maxSupply = 0;
  tokenData.freezeDefault = false;

  IHederaTokenService.TokenKey[] memory keys = new IHederaTokenService.TokenKey[](1);
  keys[0] = IHederaTokenService.TokenKey({
    keyType: 16,
    key: IHederaTokenService.KeyValue({
      inheritAccountKey: false,
      contractId: address(this),
      ed25519: bytes(''),
```

```
        ECDSA_secp256k1: bytes(''),
        delegatableContractId: address(0)
      })
    });
    tokenData.tokenKeys = keys;

    IHederaTokenService.Expiry memory expiryInfo;
    expiryInfo.autoRenewAccount = _autoRenewAccount;
    expiryInfo.second = int64(int256(block.timestamp + 90 days));
    expiryInfo.autoRenewPeriod = int64(30 days);
    tokenData.expiry = expiryInfo;

    (bool success, bytes memory result) = HTS_PRECOMPILE.call{value: msg.value}(
      abi.encodeWithSelector(
        IHederaTokenService.createFungibleToken.selector,
        tokenData,
        int64(0),
        int32(8)
      )
    );
    require(success, 'createFungibleToken call failed');
    (int64 responseCode, address createdToken) = abi.decode(result, (int64, address));
    require(responseCode == HAPI_SUCCESS, 'xBONZO token creation failed');

    xBonzoToken = createdToken;
    return createdToken;
  }
```

The `autoRenewAccount` is an address designated to fund the token's automatic renewal after it reaches its expiration date. Essentially, if the token is set to have a periodic renewal period (in this case, every 30 days), the `autoRenewAccount` will be charged the renewal fees needed to keep the token active. A malicious actor could set an arbitrary address as the `autoRenewAccount`, gaining control over the token's renewal process and preventing legitimate renewal by refusing to fund it.
This could lead to the expiration and deactivation of the xBonzo token, disrupting the entire staking system.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

It is recommended to restrict the `createXBonzoToken` function to authorized accounts, to prevent unauthorized or malicious token creation. Alternatively, consider setting the `autoRenewAccount` address during contract initialization to avoid passing this critical parameter through an unrestricted external function.

Remediation Comment

SOLVED: The `createXBonzoToken` function to was restricted only to admin account.

Remediation Hash

https://github.com/Bonzo-Labs/bonzo-staking-module/commit/5b54b86216d9645907b59ea1065e13b40ac76f6f

7.2 (HAL-02) INSUFFICIENT BONZO FUNDING FOR ACCRUED REWARDS MIGHT CAUSE UNSTAKE FAILURES

// LOW

Description

In the `BonzoStaking` contract, the function `_updateRewards` artificially increments `totalBonzoStaked` by a computed `reward` amount without actually transferring or minting the corresponding Bonzo tokens into the contract. As a result, the contract's internal accounting shows more Bonzo tokens available for staking rewards than it physically holds in its balance.

```
function _updateRewards() internal {
  uint256 timeElapsed = block.timestamp - lastUpdateTime;
  if (timeElapsed > 0 && totalBonzoStaked > 0) {
    uint256 reward = (totalBonzoStaked * annualRateBPS * timeElapsed) / (10000 * 365 days);
    if (reward > 0) {
      rewardsAllocated += reward;
      totalBonzoStaked += reward;
    }
  }
  lastUpdateTime = block.timestamp;
}
```

When a user later attempts to unstake (calling the `unstake` function), the contract will attempt to transfer the full "virtual" reward amount from itself to the user. If the contract does not actually possess enough Bonzo tokens to cover this combined principal and rewards, the Hedera Token Service transfer reverts, preventing the user from withdrawing.

```
function unstake(uint256 xAmount) external whenNotPaused nonReentrant {
  require(xAmount > 0, 'xAmount must be > 0');
  require(xAmount <= totalXBonzoSupply, 'Invalid amount');
  require(xAmount >= MIN_STAKE_UNSTAKE_AMOUNT, 'Amount too small'); // Avoid precision loss due
  _updateRewards();
```

```
uint256 redeemable = (xAmount * totalBonzoStaked) / totalXBonzoSupply;

totalBonzoStaked -= redeemable;
totalXBonzoSupply -= xAmount;

// Transfer xBONZO tokens from the user to this contract before burning.
(bool transferXSuccess, bytes memory transferXResult) = HTS_PRECOMPILE.call(
  abi.encodeWithSelector(
    IHederaTokenService.transferToken.selector,
    xBonzoToken,
    msg.sender,
    address(this),
    uint64(xAmount)
  )
);
int64 transferXResponse = transferXSuccess
  ? abi.decode(transferXResult, (int64))
  : PRECOMPILE_BIND_ERROR;
require(transferXResponse == HAPI_SUCCESS, 'xBONZO transfer to contract failed');

// Burn the transferred xBONZO tokens.
_burnHTS(xBonzoToken, uint64(xAmount));

// Transfer redeemable BONZO tokens from this contract to the user.
(bool success, bytes memory result) = HTS_PRECOMPILE.call(
  abi.encodeWithSelector(
    IHederaTokenService.transferToken.selector,
    bonzoToken,
    address(this),
    msg.sender,
    uint64(redeemable)
```

```
    )
  );
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:L/Y:L (3.1)

Recommendation

It is recommended to ensure that the contract always holds enough Bonzo tokens to cover both staked amounts and accrued rewards.

- Pre-fund the contract with sufficient Bonzo tokens to cover expected rewards.
- Automatically transfer Bonzo tokens to cover rewards during the unstake operation if the contract does not hold a sufficient balance.

Remediation Comment

ACKNOWLEDGED: The `Bonzo` team will pre-fund the contract with Bonzo tokens and keep topping them up every few months.

7.3 (HAL-03) UNLOCKED PRAGMA COMPILER

// INFORMATIONAL

Description

The file in scope currently uses floating pragma version `^0.8.11`, which means that the code can be compiled by any compiler version that is greater than or equal to `0.8.0`, and less than `0.9.0`.

It is recommended that contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

Additionally, using a newer compiler version that introduces default optimizations, including unchecked overflow for gas efficiency, presents an opportunity for further optimization.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to lock the pragma version to the same version used during development and testing. Use the up-to-date version without known vulnerabilities.

Remediation Comment

SOLVED: Pragma was locked to the 0.8.11 version.

Remediation Hash

https://github.com/Bonzo-Labs/bonzo-staking-module/commit/5b54b86216d9645907b59ea1065e13b40ac76f6f

References

Invalid GitHub URL

7.4 (HAL-04) INCORRECT ORDER OF MODIFIERS: NONREENTRANT SHOULD PRECEDE ALL OTHER MODIFIERS

// INFORMATIONAL

Description

To mitigate the risk of reentrancy attacks, a modifier named `nonReentrant` is commonly used. This modifier acts as a lock, ensuring that a function cannot be called recursively while it is still in execution. A typical implementation of the `nonReentrant` modifier locks the function at the beginning and unlocks it at the end. However, it is vital to place the `nonReentrant` modifier before all other modifiers in a function. Placing it first ensures that all other modifiers cannot bypass the reentrancy protection. In the current implementation, some functions use other modifiers before `nonReentrant`, which may compromise the protection it provides.

During the audit it was identified that the following functions does not use `nonReentrant` as a first modifier:

- `stake`
- `unstake`

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to follow the best practice of placing the `nonReentrant` modifier before all other modifiers. By doing so, one can reduce the risk of reentrancy-related vulnerabilities. This simple yet effective approach can help enhance the security posture of any Solidity smart contract.

Remediation Comment

SOLVED: The `Bonzo` team solved the issue as recommended.

Remediation Hash

https://github.com/Bonzo-Labs/bonzo-staking-module/commit/5b54b86216d9645907b59ea1065e13b40ac76f6f

7.5 (HAL-05) DUPLICATE EXCHANGE RATE FUNCTIONS LEAD TO REDUNDANT CODE

// INFORMATIONAL

Description

In the **BonzoStaking** contract, there are two public view functions - `currentExchangeRate` and `getBonzoToXBonzoRatio –` which return exactly the same value. Both compute the ratio of `totalBonzoStaked` to `totalXBonzoSupply` scaled by `PRECISION`.

Since both functions produce the same output, having two separate functions introduces minor redundancy in the codebase.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to remove the redundancy. If both functions are intended to report the same ratio, remove one of them to avoid confusion.

Remediation Comment

SOLVED: The `currentExchangeRate` function was removed from the contract.

Remediation Hash

https://github.com/Bonzo-Labs/bonzo-staking-module/commit/5b54b86216d9645907b59ea1065e13b40ac76f6f

---