SMART CONTRACTS REVIEW

zokyo

November 4th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passed a security audit.

ZOKYO AUDIT SCORING HEURIST

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 2 High issues: 2 resolved = 0 points deducted
- 4 Medium issues: 4 acknowledged = - 8 points deducted
- 10 Low issues: 4 resolved and 6 acknowledged = - 6 points deducted
- 1 Informational issue: 1 acknowledged = 0 points deducted

Thus, 100 - 8 - 6 = 86

TECHNICAL SUMMARY

This document outlines the overall security of the Heurist smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Heurist smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Heurist team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

Auditing Strategy and Techniques Applied 5

Executive Summary 7

Structure and Organization of the Document 8

Complete Analysis 9

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Heurist repository:
Repo: https://github.com/heurist-network/tge-contracts

Last commit - e0dfaacc1ffcff314e31ebe51415efb47129d713

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- /chestRewards/src/ChestRewards.sol
- ./heu-stHeu/src/StHEU.sol
- ./heu-stHeu/src/HEU.sol
- ./heu-stHeu/src/interfaces/IHEU.sol
- ./heu-stHeu/src/interfaces/IStHEU.sol
- ./heu-stHeu/src/interfaces/IStHEUEvents.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Heurist smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The Audit covered Contracts like ChestRewards, StHeu and heuristGenesis.

The `ChestRewards` contract is designed for a reward distribution system, supporting two ERC-20 tokens, HEU (silver) and stHEU (gold). Utilizing OpenZeppelin's libraries for secure token handling and access control, it allows the `operator` to allocate and distribute rewards, while users can claim accumulated tokens securely. With reentrancy protection, safe transfer methods, and detailed event logging, the contract emphasizes security and transparency. Custom errors optimize gas usage and provide clarity in cases of failure. This system is well-suited for gamified or loyalty applications, enabling efficient batch reward distribution across a large user base.

The `StHEU` contract represents a staked version of the HEU token, designed to facilitate staking, vesting, and claiming of HEU tokens in a secure and controlled manner. leveraging OpenZeppelin's ERC-20, Ownable, Pausable, and ReentrancyGuard modules, the contract ensures efficient and safe token handling. Users can lock HEU tokens to mint `stHEU` tokens, with a flexible vesting period allowing tokens to be vested and later claimed as HEU, or canceled if necessary. With built-in migration support, an adjustable vesting period, and an internal HEU/stHEU exchange rate mechanism, this contract offers a structured and flexible staking system for HEU token holders.

The `HeuristGenesis` contract is an ERC721 token smart contract designed for a limited 1,600 token minting, featuring distinct phases for whitelisting and public minting. The contract integrates OpenZeppelin modules, including `ERC721Enumerable, Ownable`, and `ReentrancyGuard`, enhancing secure and efficient token operations. The contract includes a minting fee mechanism, signature-based whitelist verification, and a two-phase sale mode (whitelist and public) managed by the owner. It also incorporates transfer control, which allows the owner to enable or disable token transfers. Error handling is extensive, covering minting constraints, signature validity, and the fee management system.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Heurist team and the Heurist team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	ExchangeRate manipulation via Donation	High	Resolved
2	Vesting During Migration Mode	High	Resolved
3	Exchange Rate Manipulation via Direct Transfers	Medium	Acknowledged
4	Zero HEU Payouts Due to Rounding in Exchange Rate Calculation	Medium	Acknowledged
5	Missing Upper Bound on vestPeriod	Medium	Acknowledged
6	Risk of Centralisation	Medium	Acknowledged
7	Insufficient HEU Tokens for Claims	Low	Acknowledged
8	Claim During Migration Mode	Low	Acknowledged
9	Missing Validation for Empty Address Inputs (ChestRewards.addRewards() and distribute())	Low	Acknowledged
10	Missing Invariant for Consistent Token Balances (ChestRewards.addRewards())	Low	Acknowledged
11	Missing Validation for Claiming More than Available Balance (ChestRewards.claimRewards())	Low	Acknowledged
12	Lack of Event Emission for Minting	Low	Resolved
13	Lack of Two-Step Ownership Transfer	Low	Acknowledged
14	Owner can renounce ownership	Low	Resolved
15	NonReentrant Modifier Not Positioned First	Low	Resolved
16	Missing Event Emission in setOperator	Low	Resolved
17	Use the proxy pattern at the early stage of the protocol	Informational	Acknowledged

HIGH-1 | RESOLVED

ExchangeRate manipulation via Donation

The StHEU.sol `donate(uint256 amount)` function allows the contract owner to donate HEU tokens to the contract. The donated tokens increase the `totalHEU` balance, which affects the exchange rate.

Manipulation Scenario:

- The contract owner (or someone with control over the owner account) could donate a large number of HEU tokens to the contract, thereby increasing `totalHEU` and boosting the exchange rate.
- With the exchange rate now inflated, the owner could claim previously vested stHEU tokens, receiving more HEU tokens than the value of the stHEU that was originally vested.

Since the `donate()` function can only be called by the owner, there is a centralization risk where the owner could manipulate the exchange rate to their benefit.

Recommendation:

Restrict the `donate()` function to only allow donations when `migrationMode` is active, to avoid affecting the exchange rate during normal operations.
Comment: The donate() function has been removed from the contract.

HIGH-2 | RESOLVED

Vesting During Migration Mode

The StHEU.sol `vest()` function does not check if `migrationMode` is enabled, allowing users to continue vesting stHEU tokens even when the contract is in migration mode.

A malicious user could vest stHEU tokens during migration, potentially leading to inconsistencies during migration. For instance, the system may not accurately account for these new vests, leading to errors or a loss of stHEU or HEU.

Recommendation:

Add a `require(!migrationMode, "Migration mode active");` check in the `vest()` function to prevent new vests during migration.

Exchange Rate Manipulation via Direct Transfers

In the StHEU the `_exchangeRate()` calculation relies on the balance of HEU tokens held by the contract (`totalHEU`). The contract does not differentiate between HEU tokens locked through the `lock()` function and tokens directly sent to the contract address.

Exchange Rate = totalHEU x 1e18 / totalSupply

- **Manipulation Scenario:**
- An attacker could directly **send HEU tokens** to the contract address without calling the `lock()` function.
- This would **increase** `totalHEU` while `totalSupply` of stHEU remains unchanged, leading to a **higher exchange rate**.
- A higher exchange rate means that future users calling `lock()` will receive **fewer stHEU tokens** for the same amount of HEU.
- Conversely, an attacker could artificially **inflate the HEU** balance before calling `claim()`, allowing them to receive **more HEU for their stHEU tokens** than they otherwise would have

Recommendation:
Maintain an internal variable `(totalLockedHEU)` that tracks HEU locked through the `lock()` function, rather than relying on `heu.balanceOf(address(this))`. Use `totalLockedHEU` in the `_exchangeRate()` calculation to accurately reflect only the HEU that has been locked by users.

Client comment: The HEU balance in StHEU contract should be always equal or lager than stHEU total supply, and we expected exchange rate to be updated whenever there is HEU transfer to stHEU contract. And the Donate is unnecessary, so this function is removed from the latest contract.
The hypothetical "attacker" won't extract values from such a manipulation and if such a manipulation wouldn't cause financial loss to the protocol (or stakers).

Zero HEU Payouts Due to Rounding in Exchange Rate Calculation

The exchange rate calculation in the `StHEU` contract is performed using the following formula:

```
return totalHEU * 1e18 / totalSupply;
```

This formula introduces a critical issue when the `totalSupply` of stHEU tokens is greater than the `totalHEU` balance in the contract. Due to Solidity's truncating division (which rounds down), small discrepancies between `totalHEU` and `totalSupply` can result in the exchange rate being rounded down to zero. In such cases, users attempting to claim their HEU tokens may receive zero HEU, even though they are entitled to a portion of the remaining balance.

For example, if `totalHEU` is 1 and `totalSupply` is greater than 1, the exchange rate will be calculated as zero due to rounding, effectively denying users any HEU on claims.

Recommendation:
To mitigate this issue, ensure that the exchange rate calculation accounts for rounding and provides users with at least a proportional share of the HEU balance, even when the supply of stHEU tokens is greater than the contract's HEU balance. A potential solution could involve implementing a minimum exchange rate to avoid rounding down to zero

Client comment: The HEU balance in StHEU contract should be always equal or lager than stHEU total supply, and we expected exchange rate to be updated whenever there is HEU transfer to stHEU contract. And the Donate is unecessary, so this function is removed from the latest contract.
If totalSupply of stHEU will not exceed totalHEU, and that means this issue won't happen.

Redundant Check for Zero Address in `_transfer` Function

Description:
The `_transfer` function in the provided in nerd token contains a redundant check for the zero address. Initially, the function checks if the to address is the zero address using a require statement. However, further into the function, there's another check for the zero address within a compound `if` condition.

Recommendation:
Remove the redundant check for the zero address in the compound `if` statement. The initial `require` statement is sufficient to prevent transfers to the zero address.

Missing Upper Bound on `vestPeriod`

In the `StHEU` contract, the `setVestPeriod` function allows the contract owner to update the vesting period via `setVestPeriod`
While the function ensures that `newPeriod` is non-zero, there is no upper bound on the value that can be set for the vesting period. This lack of an upper bound introduces a centralization risk, where the contract owner could set an unreasonably long vesting period (e.g., 100 years or more), effectively locking users' stHEU tokens indefinitely and preventing them from claiming their HEU tokens.

The absence of an upper limit on the vesting period gives the owner significant control over users' ability to claim tokens. If the owner sets an excessively long vesting period, users may be unable to access their HEU tokens for an extended time, potentially leading to a loss of liquidity and usability.

Recommendation:
Introduce a reasonable upper bound on the vesting period to prevent abuse and ensure that users can eventually claim their tokens.

MEDIUM-4 | ACKNOWLEDGED

Risk of Centralisation

The current implementation grants significant control to the owner through multiple functions that can alter the contract's state and behavior. This centralization places considerable trust in a single entity, increasing the risk of potential misuse.

If the owner's private key is compromised, an attacker could execute any function accessible to the owner, potentially leading to fund loss, contract manipulation, or service disruption.

Recommendation:
To enhance security and reduce the risk of a single point of failure, it is recommended to implement a multi-signature wallet for executing owner functions.

Client comment: For this risk, we will use multi-sig to execute owner functions.

LOW-1 | ACKNOWLEDGED

Insufficient HEU Tokens for Claims

Description: The `claim()` function does not validate whether the contract has enough HEU tokens to fulfill a user's claim. If the balance of HEU is insufficient (e.g., due to previous withdrawals or unexpected transfers), the claim could fail.

Users trying to claim their vested HEU may experience transaction reverts

Recommendation:
Add a validation to ensure that the contract holds enough HEU tokens to fulfill the claim:

```
require(heu.balanceOf(address(this)) >= heuAmount, "Insufficient HEU balance in contract to fulfill claim");
```

LOW-2 | ACKNOWLEDGED

Claim During Migration Mode

Description: Users are still able to claim their vested tokens during migration mode, which might not be intended as it can disrupt ongoing migration activities.

Allowing claims during migration can lead to potential inconsistencies, especially when trying to migrate to a new version of the contract.

Recommendation:
Prevent claims during migration mode by adding a validation:

```
require(!migrationMode, "Migration mode active, claiming is disabled");
```

LOW-3 | ACKNOWLEDGED

Missing Validation for Empty Address Inputs (`ChestRewards.addRewards()` and `distribute()`)

Description: In the `ChestRewards` contract, the `addRewards()` and `distribute()` functions do not validate whether the `rewardee` address is a non-zero address. Adding rewards or distributing them to a zero address will lock the tokens.

Scenario: The operator accidentally calls `addRewards(address(0), 100, RewardType.SILVER)`, which results in the rewards being assigned to the zero address, effectively locking the tokens and making them unclaimable.

Recommendation:
Add a validation check in `addRewards()`, `batchAddRewards()`, `distribute()`, and `batchDistribute()` to ensure that the rewardee is not a zero address (`require(rewardee != address(0), "Invalid rewardee address")`).

LOW-4 | ACKNOWLEDGED

Missing Invariant for Consistent Token Balances (`ChestRewards.addRewards()`)

Description: In the `ChestRewards` contract, the `addRewards()` function does not maintain an invariant to ensure that the **total assigned rewards** do not exceed the **contract balance** for `silverRewardToken` or `goldRewardToken`. There is no guarantee that sufficient tokens exist in the contract to cover all assigned rewards.

Scenario: The operator adds rewards without ensuring the corresponding tokens are transferred to the contract, resulting in users being unable to claim their rewards due to insufficient balance.

Recommendation:
Introduce a check during reward addition to ensure that the **total rewards assigned** are always less than or equal to the contract's **token balance**. Alternatively, require operators to deposit the corresponding reward tokens when adding rewards

LOW-5 | ACKNOWLEDGED

Missing Validation for Claiming More than Available Balance (`ChestRewards.claimRewards()`)

Description: In the `ChestRewards` contract, the `claimRewards()` function does not verify that the contract has enough balance to fulfill the user's reward claim. The function calls `safeTransfer()` directly, which may fail if the contract holds insufficient tokens.

Scenario: Multiple users attempt to claim their rewards, but the contract's balance is insufficient. Users experience transaction failures and incur gas fees without receiving any rewards.

Recommendation:
Before executing `safeTransfer()`, add a check to ensure the contract holds enough tokens to fulfill the claim
`(require(goldRewardToken.balanceOf(address(this)) >= rewards, "Not enough tokens to fulfill claim")).`

LOW-6 | RESOLVED

Lack of Event Emission for Minting

Description: In HEU.sol `mint()` function does not emit a custom event indicating that new tokens have been minted. While the ERC20 standard emits a `Transfer` event for minting, adding a custom event would make it easier to track and audit minting operations specifically.

Scenario: If large minting operations occur, users, auditors, and other stakeholders may have difficulty distinguishing normal token transfers from minting operations without detailed blockchain analysis.

Recommendation:
Emit a custom `Mint` event in the `mint()` function to improve transparency. This would allow easier monitoring and auditing of minting operations.

LOW-7 | ACKNOWLEDGED

Lack of Two-Step Ownership Transfer

The contracts does not implement a two-step process for transferring ownership. In its current state, ownership can be transferred in a single step, which can be risky as it could lead to accidental or malicious transfers of ownership without proper verification.

Recommendation:
Implement a two-step process for ownership transfer where the new owner must explicitly accept the ownership. It is advisable to use OpenZeppelin's Ownable2Step.

LOW-8 | RESOLVED

Owner can renounce ownership

The `Ownable` contracts includes a function named `renounceOwnership()` which can be used to remove the ownership of the contract.
If this function is called on the `StHEU, HEU` contract, it will result in the contract becoming disowned. This would subsequently break functions of the token that rely on `onlyOwner` modifier.

Recommendation:
override the function to disable its functionality, ensuring the contract cannot be disowned e.g.

LOW-9 | RESOLVED

NonReentrant Modifier Not Positioned First

The `nonReentrant` modifier should be applied as the first modifier in any function to prevent reentrancy attacks. Placing it after other modifiers may leave the function vulnerable, as external calls or conditions evaluated by earlier modifiers can still be subject to reentrancy exploits.

Recommendation:
Reorder the `nonReentrant` modifier to be the first modifier in all functions where it's applied.

LOW-10 | RESOLVED

Missing Event Emission in setOperator

The `setOperator` function in ChestRewards allows the owner to change the operator address:

However, the function does not emit an event to log the change of the operator. Event emissions are crucial for tracking state changes on-chain, providing transparency and facilitating easier auditing of contract activities.

Recommendation:
To improve the contract's transparency and auditability, it is recommended to emit an event whenever the operator is changed.

INFORMATIONAL-1 | ACKNOWLEDGED

Use the proxy pattern at the early stage of the protocol

As the protocol evolves, incorporating upgradeability becomes essential for managing potential vulnerabilities, adding new features, and ensuring long-term viability. It is advisable to implement the proxy pattern at the early stage of the protocol, as recommended by ZkSync in their best practices documentation.

Impact: Utilizing the proxy pattern allows for more flexibility in contract upgrades without losing state or requiring users to migrate to new contracts. This can significantly enhance the governance and adaptability of the protocol as it grows and evolves.

Recommendation:
Implement the proxy pattern early in the development phase. This will enable seamless upgrades and facilitate future enhancements while maintaining user trust and minimizing disruptions. For more details, refer to the ZkSync documentation: ZkSync Best Practices.

	/chestRewards/src/ChestRewards.sol **./heu-stHeu/src/StHEU.sol** **./heu-stHeu/src/HEU.sol** **./heu-stHeu/src/interfaces/IHEU.sol** **./heu-stHeu/src/interfaces/IStHEU.sol** **./heu-stHeu/src/interfaces/IStHEUEvents.sol**
Reentrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Heurist team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Heurist team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo