SMART CONTRACTS REVIEW

zokyo

August 1st 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.

HYPOTHETICAL 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
- 4 High issues: 2 resolved and 2 acknowledged = - 8 points deducted
- 1 Medium issue: 1 resolved = 0 points deducted
- 6 Low issues: 4 resolved and 2 acknowledged = - 2 points deducted
- 7 Informational issues: 5 resolved and 2 acknowledged = 0 points deducted

Thus, 100 - 8 - 2 = 90

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

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

Last commit - cc62d08

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

- ZkImagine.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 Zokyo team has performed a security audit of the provided codebase. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

ZkImagine is an ERC721-based non-fungible token (NFT) smart contract designed for deployment on the zkSync network. This contract includes features such as referral-based minting discounts, partner NFT whitelisting for free mints, signature-based free minting, and secure fee handling.

The ZkImagine smart contract enables standard minting where users can mint new NFTs by paying a mint fee of `0.0006` ether. A referral system allows referrers to earn a percentage of the mint fee as a reward, while minters receive a discount when using a referral. The minting process is handled by the `mint()` function, which ensures that the referral fee is correctly allocated and the required minting fee is paid.

The partner whitelist feature allows holders of whitelisted partner NFTs to mint NFTs for free once per day. Partner NFTs must be whitelisted by the contract owner using the `addWhitelistedNFT()` function. The `partnerFreeMint()` function checks the ownership of the partner NFT and ensures that the daily minting limit is respected.

The signature-based free minting feature enables the contract owner to authorize free minting via signatures. The `signatureFreeMint()` function validates the signature and ensures that the same signature cannot be reused within a day.

The contract supports dynamic base URI for token metadata, allowing the owner to update the base URI using the `setBaseURI()` function. This flexibility ensures that the metadata can be managed and updated as needed, keeping the platform up-to-date with the latest standards and practices. The `tokenURI()` function returns the full URI for the token metadata for a specified token ID, ensuring that the metadata can be easily accessed.

The referral system in ZkImagine allows users to earn referral fees when others mint NFTs using their referral address. Referral fees accumulate and can be claimed using the `claimReferralFee()` function. This incentivizes users to promote the platform, driving user engagement and growth. Additionally, the contract owner can claim the collected minting fees using the `claimFee()` function, which ensures that referral fees are preserved for later claims.

The ZkImagine contract emits several events to log key activities, such as `WhitelistedNFTAdded` and `WhitelistedNFTRemoved` for managing partner NFTs, `Minted`, `PartnerFreeMint`, and `SignatureFreeMint` for different minting activities, and `ReferralFeeClaimed` and `FeeClaimed` for claiming fees. These events provide transparency and traceability, enabling users and developers to track the contract's operations and interactions.

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	Unrestricted repeated minting via Partner NFT Transfer Exploit	High	Resolved
2	Reentrance in partnerFreeMint	High	Resolved
3	Insufficient Validation of Referral Address in mint	High	Acknowledged
4	Holder of Partner NFT can mint unlimited NFTs in partnerFreeMint	High	Acknowledged
5	Missing gap to Avoid Storage Collisions	Medium	Resolved
6	Use of ecrecover Instead of Native Account Abstraction in _recoverSigner Function	Low	Resolved
7	Lack of Two-Step Ownership Transfer	Low	Acknowledged
8	Owner can renounce ownership	Low	Resolved
9	Missing Event Emission in setBaseURI	Low	Resolved
10	Centralization Risk	Low	Acknowledged
11	Gas Intensive While Loop in _updateGlobalTimeThreshold Function	Low	Resolved
12	Immutable Mint Fee and Referral Discount Percentage	Informational	Resolved
13	Signature Replay Vulnerability Across Different Domains in Function signatureFreeMint	Informational	Acknowledged
14	Token ID potential duplication due to manipulable total supply	Informational	Resolved

#	Title	Risk	Status
15	Missing Boundaries Check in Setter Functions	Informational	Acknowledged
16	Missing Sanity Checks in initialize Function	Informational	Resolved
17	Unused Import of Ownable	Informational	Resolved
18	Missing nonReentrant Modifier	Informational	Resolved

HIGH-1 | RESOLVED

Unrestricted repeated minting via Partner NFT Transfer Exploit

The `partnerFreeMint` function is vulnerable to an exploit where users can repeatedly transfer the partner NFT to different wallets, allowing unlimited minting without adhering to the time restrictions. This occurs because the contract does not verify the ID of the partner NFT, making it possible for users to undermine the intended limitations.

Recommendation:
Implement a mechanism to track the usage of each specific partner NFT ID for the `partnerFreeMint` function. This can be achieved by maintaining a mapping of NFT IDs to their last minting timestamp or a boolean indicating whether they have been used for minting. Additionally, ensure that users cannot reuse the same NFT for minting until the time restriction has elapsed.

HIGH-2 | RESOLVED

Reentrance in `partnerFreeMint`

The `partnerFreeMint` function doesn't implement the checks-effects-interactions pattern. The `_safeMint` function call occurs before updating the `lastMinted` mapping. If the `to` address is a contract, it could call back into the `partnerFreeMint` function, allowing re-entrance before `lastMinted` is updated and bypassing the 24 hours limit.

Recommendation:
To prevent reentrancy attacks, update the `lastMinted` mapping before calling the `_safeMint` function.

Fix: Client addressed the issue in commit a27af000 by adhering to the checks-effects-interactions pattern. It is recommended though to add an extra layer of security by applying the reentrancy guard modifier to the function since the code base utilizes it already in other places of the codebase.

Update: The client added the reentrancy guard modifier.

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.

HIGH-3 | ACKNOWLEDGED

Insufficient Validation of Referral Address in `mint`

The `mint` function allows users to specify a referral address to receive a discount on the minting fee. However, the referral address is not validated. This means that users can set the referral address to any arbitrary address, including an address that they control, to exploit the discount mechanism and potentially drain referral rewards.

Recommendation:
To prevent abuse of the referral system, implement validation checks to ensure that the referral address is a legitimate and authorized referrer.

HIGH-4 | ACKNOWLEDGED

Holder of Partner NFT can mint unlimited NFTs in partnerFreeMint

The `partnerFreeMint` function allows users to mint a new token if they hold at least one token of a whitelisted partner NFT. However, this can be exploited by an attacker who holds a single partner NFT, allowing them to mint an unlimited number of `ZkImagine` NFTs over time. This undermines the intended scarcity and value of the NFTs.

Recommendation:
Limit the number of free mints a single partner NFT holder can perform.

Auditor's Comment:
In the snippet below:

```
_updateGlobalTimeThreshold();
nextMint[partnerNFTAddress][partnerNFTtokenId] = globalTimeThreshold;
```

The new value of `globalTimeThreshold` becomes `~ block.timestamp`.
Theoretically, The attacker can mint on every block throughout the day.
The issue also exists in `signatureFreeMint` as the client decided to make the signature replayable :

```
_updateGlobalTimeThreshold();
nextSignatureMint[signature] = globalTimeThreshold;
```

But This basically makes it replayable without a limit.

MEDIUM-1 | RESOLVED

Missing `gap` to Avoid Storage Collisions

The `ZkImagine` contract is intended to be an upgradeable smart contract, but do not have a __gap variable.

In upgradeable contracts, it's crucial to include a `_gap` to ensure that any additional storage variables added in future contract upgrades do not collide with existing storage variables. This is especially important when inheriting from multiple upgradeable contracts.

Recommendation:
Include a `_gap` as the last storage variable to `ZkImagine` contract to reserve space for future storage variables and prevent storage collisions. This is a common practice to ensure compatibility and avoid issues when upgrading the contract in the future.

LOW-1 | RESOLVED

Use of `ecrecover` Instead of Native Account Abstraction in `_recoverSigner` Function

The `_recoverSigner` function uses the `ecrecover` method for signature recovery. When deploying on zkSync, it is recommended to use native account abstraction instead of `ecrecover` for signature validation, see here.

Recommendation:
Modify the signature recovery mechanism to use zkSync's native account abstraction methods for better security and compatibility.

LOW-2 | ACKNOWLEDGED

Lack of Two-Step Ownership Transfer

The `ZkImagine` contract 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-3 | 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 `ZkImagine` contract, it will result in the contract becoming disowned. This would subsequently break functions of the ZkImagine contract that rely on `onlyOwner` modifier.

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

```
function renounceOwnership() public override onlyOwner {
revert ("renounceOwnership is disabled");
}
```

LOW-4 | RESOLVED

Missing Event Emission in `setBaseURI`

The `setBaseURI` function allows the contract owner to update the base URI for the NFTs. However, the function does not emit an event when the base URI is changed. Emitting an event for such actions is a best practice as it provides transparency and helps in tracking changes on the blockchain.

Recommendation:
Add an event emission to the `setBaseURI` function to log when the base URI is updated.

LOW-5 | ACKNOWLEDGED

Centralization Risk

The `ZkImagine` contract grants significant control to the contract owners through several functions. This centralization poses a substantial risk, as it places considerable trust and control in a single entity. If the owner's private key is compromised, it could lead to catastrophic disruptions or malicious misuse of the contract.

Recommendation:
Use a multi-signature wallet for executing Owner functions. This requires multiple authorized signatures to approve critical actions, reducing the risk of a single point of failure.

LOW-6 | RESOLVED

Gas Intensive While Loop in _updateGlobalTimeThreshold Function

The `_updateGlobalTimeThreshold` function utilizes a while loop that performs storage writes, which are among the most expensive operations in the EVM. If the loop iterates many times, it can quickly exceed the gas limit, causing the transaction to fail and undermining the contract's functionality. This while loop can be optimized by using a simple mathematical formula to achieve the same result, significantly reducing gas costs.

Recommendation:
Replace the while loop in `_updateGlobalTimeThreshold` with a simple mathematical formula to avoid excessive gas costs and potential transaction failures. The `globalTimeThreshold` can be updated in one step by calculating the smallest multiple of `freeMintCooldownWindow` that is greater than the current block timestamp.

INFORMATIONAL-1 | RESOLVED

Immutable Mint Fee and Referral Discount Percentage

The `mintFee` and `referralDiscountPct` are defined as `constant` values, making them immutable. This inflexibility can be problematic if the project needs to adjust the minting fee or referral discount percentage in the future due to changes in market conditions or other considerations.

Recommendation:
Replace the `constant` keyword with state variables and provide functions to allow the owner to update these values. Ensure that these functions are protected by appropriate access control (e.g., `onlyOwner`) to prevent unauthorized changes.

INFORMATIONAL-2 | ACKNOWLEDGED

Signature Replay Vulnerability Across Different Domains in Function `signatureFreeMint`

The current implementation of the `signatureFreeMint` function does not include domain-specific parameters in the signature, such as the contract address and chain ID. This omission allows for the same signature to be replayed across different domains, leading to potential exploitation where an attacker can mint tokens on different domains using the same signature.

Recommendation:
To mitigate this vulnerability, it is essential to incorporate domain-specific parameters into the signature. Include the contract address and chain ID as part of the data being signed. This ensures that each signature is unique to a specific domain, preventing replay attacks across different domains.

INFORMATIONAL-3 | RESOLVED

Token ID potential duplication due to manipulable total supply

The current implementation generates Token IDs based on the total supply of tokens. This approach is problematic, especially for burnable ERC721 tokens where the total supply can be decreased, potentially leading to the creation of duplicate Token IDs. When tokens are burned, the total supply reduces, causing subsequent mints to potentially reuse previously assigned Token IDs, thereby violating the uniqueness property of Token IDs.

Recommendation:
To ensure the uniqueness of Token IDs and prevent duplication, it is recommended to use a separate, dedicated counter for Token ID generation. This counter should increment with each mint and should not be influenced by current total supply of tokens or changes to the total supply. By decoupling Token ID generation from the total supply, you can maintain the integrity and uniqueness of Token IDs.

INFORMATIONAL-4 | ACKNOWLEDGED

Missing Boundaries Check in Setter Functions

The setter functions `setMintFee`, `setReferralDiscountPct`, and `setFreeMintCooldownWindow` in the `ZkImagine` contract are used to update various important parameters. While they perform some basic checks, they are missing additional boundary checks to ensure the parameters are within sensible ranges. This can potentially lead to issues if invalid values are set, affecting the contract's functionality or causing unintended behavior.

Recommendation:
Add boundary checks in the setter functions to ensure that the parameters are within appropriate ranges and prevent invalid values from being set. E.g.

```
function setMintFee(uint256 fee) external onlyOwner {
require(fee > 0, "Fee must be greater than 0");
require(fee <= 1 ether, "Fee exceeds maximum allowed"); // Example upper limit mintFee =
fee; emit MintFeeChanged(fee); }
```

INFORMATIONAL-5 | RESOLVED

Missing Sanity Checks in `initialize` Function

The `initialize` function in the `ZkImagine` contract sets initial values for various parameters, including the mint fee, referral discount, cooldown window, and start timestamp. However, it lacks sanity checks to ensure that these parameters are valid and within reasonable ranges. Without these checks, the contract may be initialized with incorrect or invalid values, E.g. the mint fee could be initialized to zero.

Recommendation:
Add sanity checks to the `initialize` function to validate the parameters and ensure they are within acceptable ranges. This will help prevent initialization with unintended or invalid values and ensure the contract operates as expected.

INFORMATIONAL-6 | RESOLVED

Unused Import of `Ownable`

Description: The contract imports `Ownable` from OpenZeppelin's contracts but does not use it in the code. Instead, it uses `OwnableUpgradeable`, which is designed for upgradeable contracts and provides similar functionality. This makes the import of `Ownable` redundant and unnecessary.

Recommendation:
Remove the unused import of `Ownable` to clean up the code and avoid unnecessary dependencies. This will make the contract more readable and reduce potential confusion regarding which ownership management implementation is being utilized.

INFORMATIONAL-7 | RESOLVED

Missing `nonReentrant` Modifier

The `ZkImagine` smart contract implements the Check-Effects-Interactions (CEI) pattern, which is a recognized best practice to prevent reentrancy attacks. This pattern ensures that state changes are made before external calls, thereby mitigating reentrancy risks. However, certain functions such as `partnerFreeMint`, `signatureFreeMint` and `claimReferralFee` do not include the `nonReentrant` modifier, which serves as an additional safeguard against reentrancy attacks. While the current implementation uses CEI, adding the `nonReentrant` modifier would provide an extra layer of security.

Recommendation:
Add the `nonReentrant` modifier to functions that involve external interactions after state changes, even if the CEI pattern is correctly implemented. This practice enhances the contract's security by preventing any possibility of reentrancy, thus offering additional assurance against potential vulnerabilities.

	ZkImagine.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