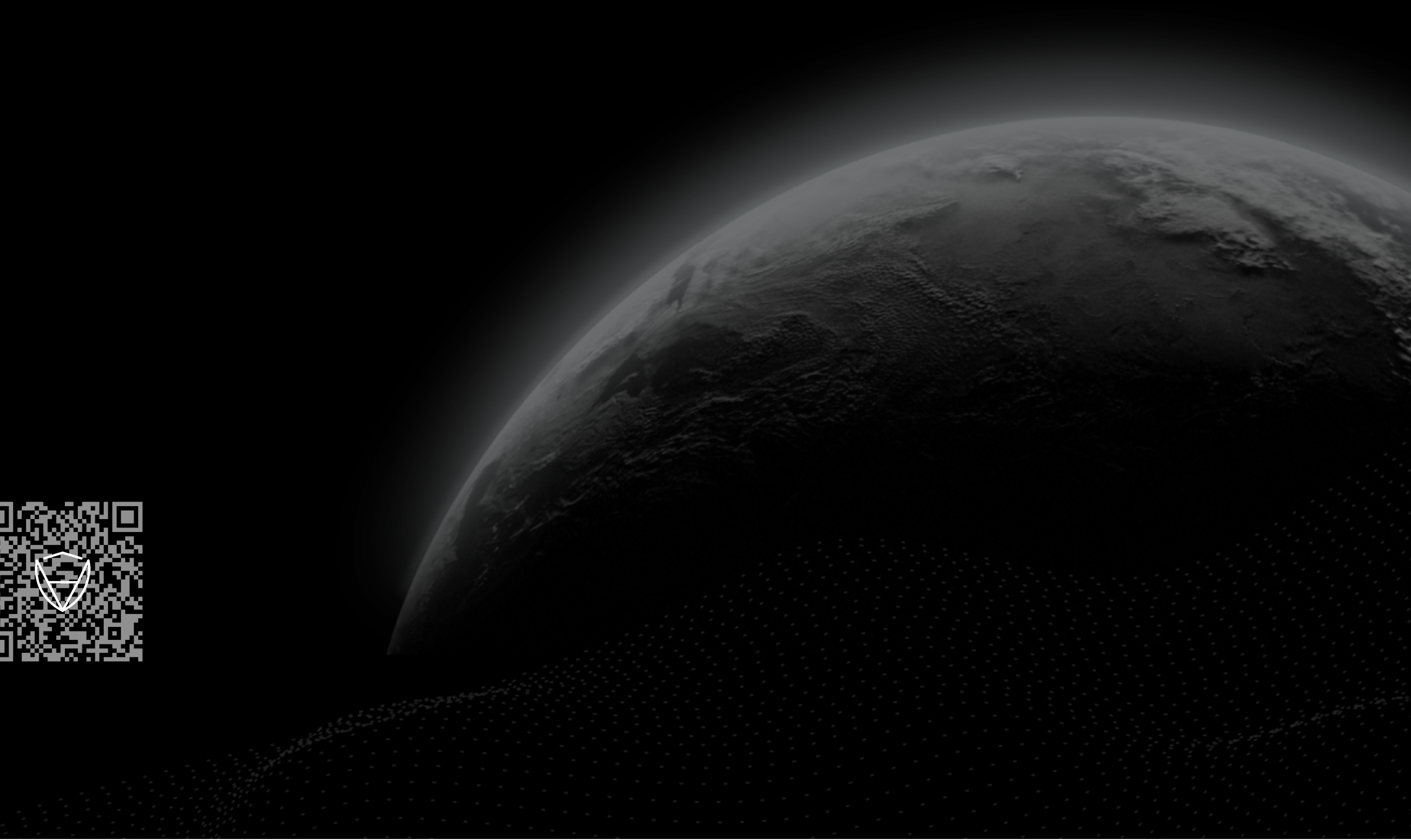
Security Assessment

RapChain-Audit

CertiK Assessed on Jan 19th, 2024



CertiK Assessed on Jan 19th, 2024

RapChain-Audit

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
DeFi	Binance Smart Chain (BSC) | Ethereum (ETH)	Formal Verification, Manual Review, Static Analysis
LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 01/19/2024	N/A

CODEBASE

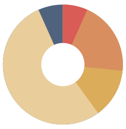
https://github.com/OurHappy/rapchain-prototype/tree/aa0665ef503fdca40ed9669b5cd7b10cc1ce6b90

View All in Codebase Page

Highlighted Centralization Risks

- Withdraws can be disabled
- Privileged role can mint tokens

Vulnerability Summary



15	10	0	0	5	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined

🟥 1	Critical	1 Acknowledged	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
🟧 3	Major	2 Resolved, 1 Acknowledged	Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
🟨 2	Medium	1 Resolved, 1 Acknowledged	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
🟨 8	Minor	6 Resolved, 2 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
🟦 1	Informational	1 Resolved	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | RAPCHAIN-AUDIT

CODEBASE | RAPCHAIN-AUDIT

Repository

https://github.com/OurHappy/rapchain-prototype/tree/aa0665ef503fdca40ed9669b5cd7b10cc1ce6b90

AUDIT SCOPE | RAPCHAIN-AUDIT

2 files audited ● 2 files with Acknowledged findings

ID	Repo	File	SHA256 Checksum
● RCO	OurHappy/rapchain-prototype	contracts/RapChain.sol	28eb0602b0c5cbab5bb7833e17809d5b9a042c4c8e87fa3604a2a2525acb255c
● RNF	OurHappy/rapchain-prototype	contracts/RapNFT.sol	f40e288ba3d2d17893ef7e47d25b3dd51affa4d74c57cb82dbbea681a3a6d2da

APPROACH & METHODS | RAPCHAIN-AUDIT

This report has been prepared for RapChain to discover issues and vulnerabilities in the source code of the RapChain-Audit project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

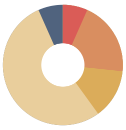
The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

FINDINGS | RAPCHAIN-AUDIT



15	1	3	2	8	1
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for RapChain-Audit. Through this audit, we have uncovered 15 issues ranging from different severity levels. Utilizing the techniques of Static Analysis & Manual Review to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
RCO-19	**Potential Honeypot Risk And Manipulation Of Winning Result**	**Centralization**	**Critical**	● **Acknowledged**
OHB-03	**Centralization Related Risks**	**Centralization**	**Major**	● **Acknowledged**
RCO-06	`pause/unpause` Functionalities Not Implemented	Logical Issue	Major	● Resolved
RCO-07	The Restarted Game Cannot Be Ended	Logical Issue	Major	● Resolved
RCO-08	Block Stuffing Attacks	Concurrency	Medium	● Acknowledged
RCO-15	Potential Signature Replay Attack	Access Control	Medium	● Resolved
OHB-02	Missing Zero Address Validation	Volatile Code	Minor	● Resolved
RCO-04	Potential Denial Of Service Caused By Buyer	Denial of Service	Minor	● Acknowledged
RCO-05	Check For "ClaimEnable" Flag When Referrer Withdraws Reward	Logical Issue	Minor	● Resolved
RCO-10	Potential Divide By Zero	Logical Issue	Minor	● Resolved
RCO-12	The `RapChain` Contract Can Be Reinitialized	Logical Issue	Minor	● Acknowledged

ID	Title	Category	Severity	Status
RCO-13	Potential Malformed NFT Token ID	Logical Issue	Minor	● Resolved
RCO-16	Check-Effects-Interactions Pattern Violation	Concurrency	Minor	● Resolved
RCO-18	The `referrer` Could Be Any Address	Access Control	Minor	● Resolved
RCO-17	Inconsistent Comment And Code	Inconsistency	Informational	● Resolved

RCO-19 POTENTIAL HONEYPOT RISK AND MANIPULATION OF WINNING RESULT

Category	Severity	Location	Status
Centralization	● **Critical**	**contracts/RapChain.sol: 375**	● **Acknowledged**

Description

In the contract `RapChain` the role `signer` has authority to sign a signature for a specific chain `id` and `len` . This signature is then used by users when calling the `buy()` function to participate in the game as a potential winner.

Any compromise to the `signer` account may allow the hacker to take advantage of this authority and sign a signature for themselves to join the game, and then end the game after 30 minutes, thus ensuring they become the winner and claim rewards.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[RapChain Team, 01/16/2024]:

1. Core Protocol Design: The reliance on server-generated signatures for verifying AI-generated content is a fundamental aspect of the RapChain protocol. This design is crucial for integrating advanced AI technology and ensuring the uniqueness and authenticity of the content used within the protocol. The offchain AI model plays a vital role in content generation, and the server signature acts as a bridge between offchain innovation and onchain trust.

2. Continuity of Operation: In the event of a signer key compromise, the security architecture of RapChain allows for the uninterrupted generation of AI-generated signatures. The protocol is designed to maintain operational resilience, ensuring that the generation of AI content and its corresponding signatures continues seamlessly. This design choice reflects a balance between innovation, user experience, and risk management.

[CertiK, 01/16/2024]:

It should be noted that the centralization risk issue still exists. CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

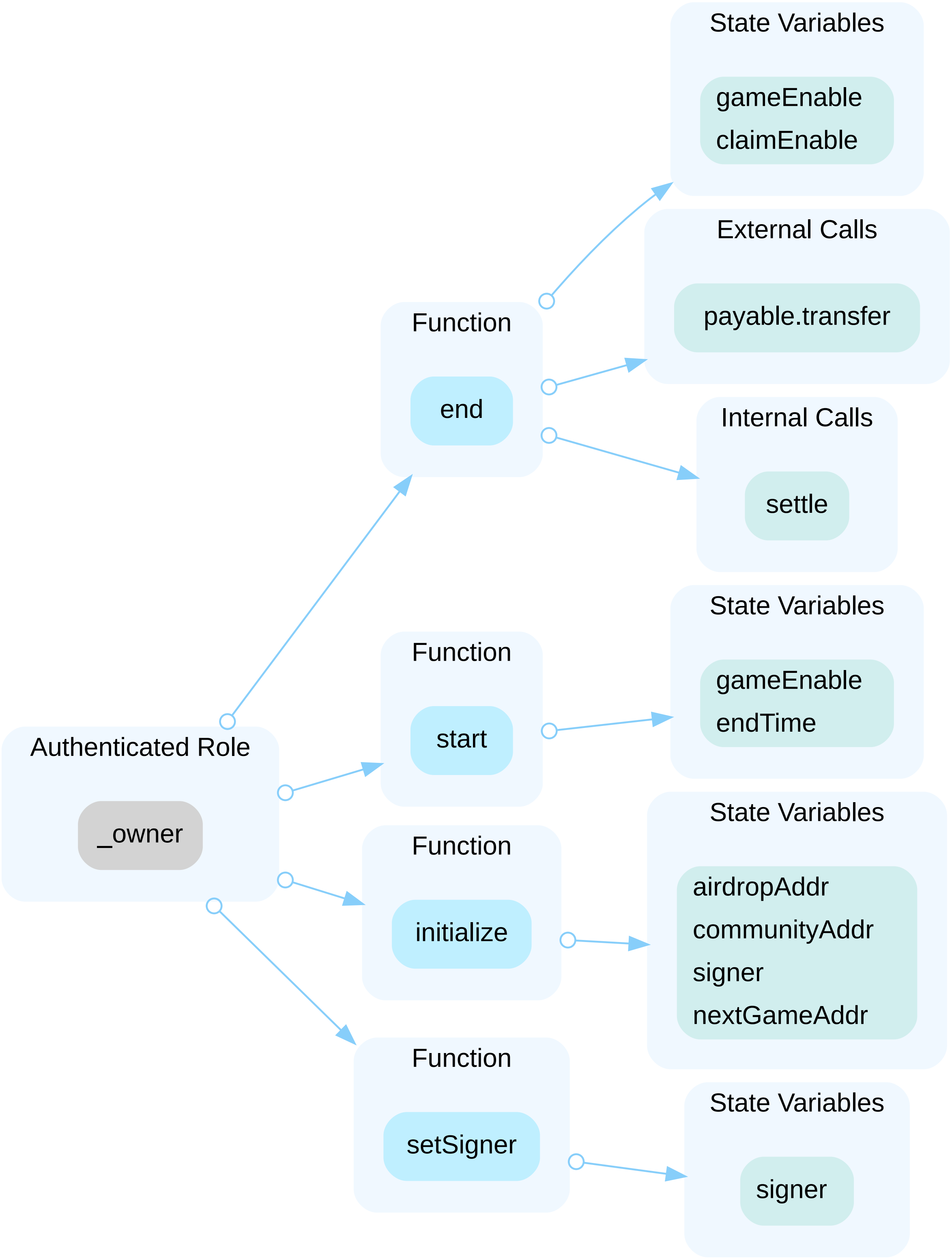
OHB-03 | CENTRALIZATION RELATED RISKS

Category	Severity	Location	Status
Centralization	● **Major**	**contracts/RapChain.sol: 122, 134, 145, 163; contracts/RapNFT.sol: 29, 37, 54**	● **Acknowledged**

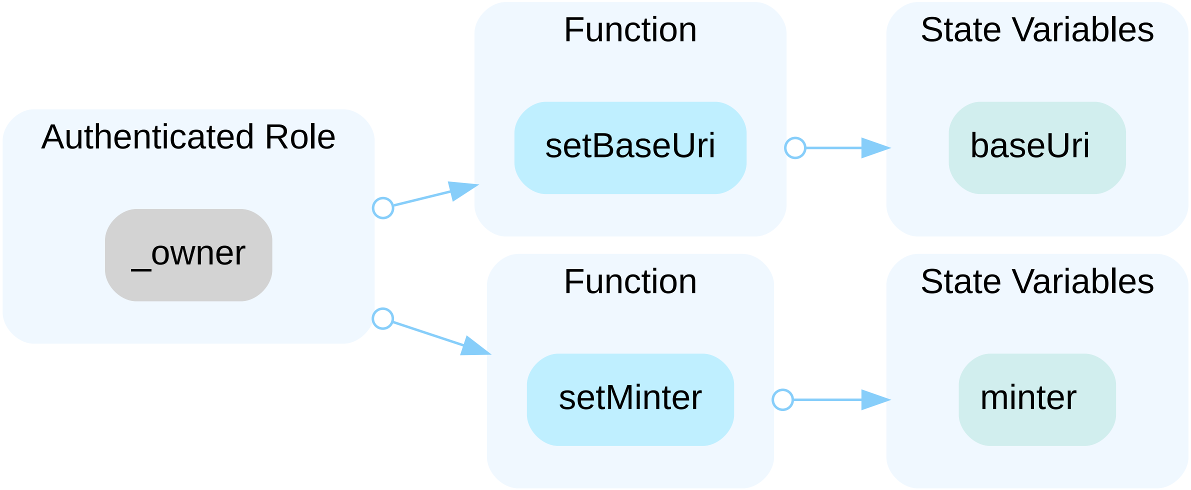
Description

In the contract `RapChain` the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and

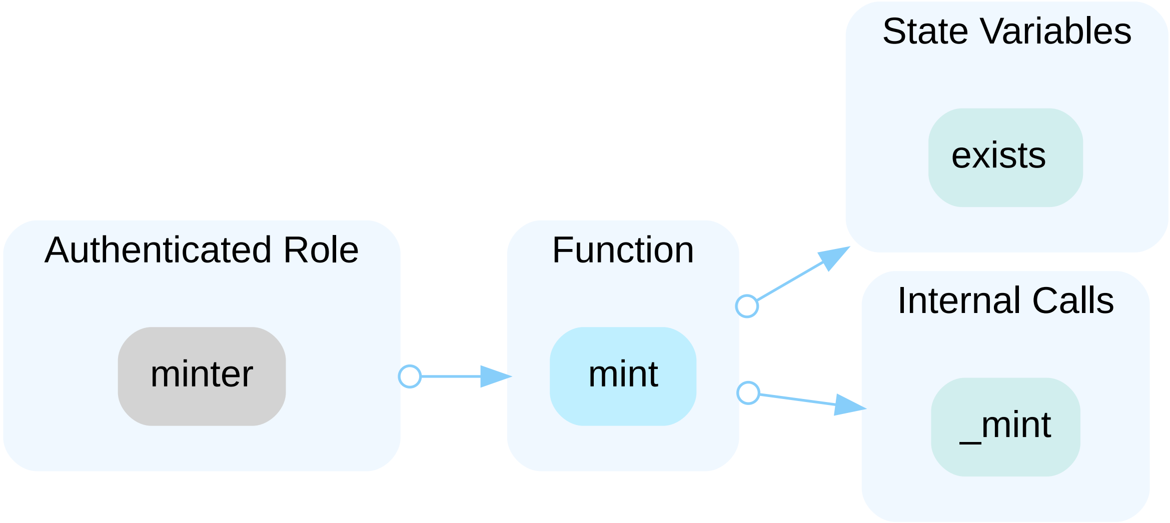
- change signer to manipulate the game winner.
- change `communityAddr` and `nextGameAddr` to receive funds.
- start or end the game.
- pause claiming rewards.



In the contract `RapNFT` the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and set minter and baseUri.



In the contract `RapNFT` the role `minter` has authority over the functions shown in the diagram below. Any compromise to the `minter` account may allow the hacker to take advantage of this authority and mint RapNFT tokens.



Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[RapChain Team, 01/16/2024]:

1. Core Protocol Design: The reliance on server-generated signatures for verifying AI-generated content is a fundamental aspect of the RapChain protocol. This design is crucial for integrating advanced AI technology and ensuring the uniqueness and authenticity of the content used within the protocol. The offchain AI model plays a vital role in content generation, and the server signature acts as a bridge between offchain innovation and onchain trust.

2. Continuity of Operation: In the event of a signer key compromise, the security architecture of RapChain allows for the uninterrupted generation of AI-generated signatures. The protocol is designed to maintain operational resilience, ensuring that the generation of AI content and its corresponding signatures continues seamlessly. This design choice reflects a balance between innovation, user experience, and risk management.

[CertiK, 01/16/2024]:

It should be noted that the centralization risk issue still exists. CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

RCO-06 | `pause/unpause` FUNCTIONALITIES NOT IMPLEMENTED

Category	Severity	Location	Status
Logical Issue	● Major	contracts/RapChain.sol: 38~39	● Resolved

Description

The contract inherits `Pausable` and uses extensively the `whenNotPaused` modifier to prevent many functions from being called if the contract is paused. However, the contract does not implement any function allowing to set `_paused` as true. Therefore, the contract cannot be paused.

Recommendation

We recommend implementing functions allowing to pause and unpause the contract.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

RCO-07 | THE RESTARTED GAME CANNOT BE ENDED

Category	Severity	Location	Status
Logical Issue	● Major	contracts/RapChain.sol: 168, 172	● Resolved

Description

The `end()` function intended to conclude the current game session. It performs the following actions:

1. Sets `gameEnable` to `false` to disable the game.
2. Sets `claimEnable` to `true` to allow players to claim their rewards.
3. Calculate and distribute rewards, which transfer 10% of the contract's funds to the `nextGameAddr`.

However, if the game is restarted by the owner, the `claimEnable` flag remains set to `true`, and due to the existing check within the `end()` function:

```
168  require(claimEnable == false, "ended");
```

The owner is unable to execute the `end()` function again to properly conclude the game. This results in a situation where the funds are locked within the contract, and players are unable to claim their rewards.

Proof of Concept

```
contract ContractTest is Test {
    address public alice = makeAddr("alice");
    RapNFT nft;
    RapChain rap;
    address signer = vm.addr(1);

    receive() payable external {}

    function setUp() public {
        nft = new RapNFT(address(this),"","");
        rap = new RapChain(address(this), address(nft));
        rap.initialize(address(this), address(this), address(this), signer);
        nft.setMinter(address(rap));
        vm.deal(alice, 1 ether);
    }

    function testRestart() public {
        rap.start();
        uint256 one = 1;
        bytes32 hash =
MessageHashUtils.toEthSignedMessageHash(keccak256(abi.encodePacked(one, one)));
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(1, hash);
        bytes memory signature = abi.encodePacked(r, s, v);
        vm.prank(alice);
        rap.buy{value: 1e18 / 2000}(signature, 1, 1, address(0));
        vm.warp(block.timestamp + 1.1 days);
        rap.end();
        // restart
        rap.start();
        vm.warp(block.timestamp + 1.1 days);
        rap.end();
    }
}
```

```
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 4.62ms

Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/Contract.t.sol:ContractTest
[FAIL. Reason: ended] testRestart() (gas: 460142)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Recommendation

Consider adding a check in the `start()` function to ensure the game cannot be restarted.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

RCO-08 | BLOCK STUFFING ATTACKS

Category	Severity	Location	Status
Concurrency	● Medium	contracts/RapChain.sol: 363	● Acknowledged

Description

The smart contract contains a game mechanism where the `endTime` is extended by 30 minutes (`endTime += 30 * 60`) each time a player purchases a rap chain. The game is set to end when `block.timestamp` exceeds `endTime` . At this point, no further purchases can be made, and the last buyer is declared the winner.

However, this mechanism is susceptible to a block stuffing attack. An attacker can exploit this by issuing a series of transactions with higher gas fees to fill the blocks' gas limits, effectively preventing the `buy()` function from being executed by other players. If the attacker manages to monopolize block space for the subsequent 30 minutes, they can ensure they are the last buyer and thus guarantee their victory.

If the winner's reward is greater than the cost, the attack is profitable.

Recommendation

To mitigate this risk, consider implementing a randomized or a fixed grace period after the `endTime` has been reached, during which transactions can still be processed to determine the final winner. Additionally, a commit-reveal scheme could be employed to prevent attackers from being certain of the game's outcome, hence disincentivizing block stuffing.

Alleviation

[RapChain Team, 01/16/2024]:

1. Game Purchase Cap: The RapChain game is designed with a cap of 100,000 purchases. This cap significantly limits the potential profit from winning the game, making it economically unfeasible for an attacker to sustain a block stuffing attack for 30 minutes. The cost of monopolizing block space for such a duration would outweigh the potential rewards from winning the game.

2. Economic Disincentive for Attackers: The financial implication of executing a block stuffing attack, combined with the capped reward structure, serves as a strong disincentive for potential attackers. The cost-benefit analysis does not favor the attacker, thereby reducing the likelihood of such an attack being attempted.

3. Robust Game Design: The RapChain game's design, including the purchase cap, reflects a balance between an engaging user experience and security considerations. This cap is an integral part of the game's strategy, encouraging fair play and competition among participants.

RCO-15 | POTENTIAL SIGNATURE REPLAY ATTACK

Category	Severity	Location	Status
Access Control	● Medium	contracts/RapChain.sol: 379	● Resolved

Description

The signed messages do not contain a domain separator. Without the domain separator allows the signature to be reused on other contracts or chains.

Scenario

1. The `RapChain` contract is deployed to addrA, and the signer issues signatures for players.
2. The game in addrA is ended.
3. The project owner deploys the `RapChain` contract to addrB with the same signer.
4. The signature that has been used for addrA can be reused in addrB.

Recommendation

Consider adding `address(this)` and `block.chainid` to the message, and using a different signer wallet if the contract is deployed multiple times.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

OHB-02 | MISSING ZERO ADDRESS VALIDATION

Category	Severity	Location	Status
Volatile Code	● Minor	contracts/RapChain.sol: 112, 123, 124, 125, 126, 135; contracts/RapNFT.sol: 30	● Resolved

Description

Addresses are not validated before assignment or external calls, potentially allowing the use of zero addresses and leading to unexpected behavior or vulnerabilities. For example, transferring tokens to a zero address can result in a permanent loss of those tokens.

```
112         rapNFT = _rapNFT;
```

- `_rapNFT` is not zero-checked before being used.

```
123         communityAddr = _community;
```

- `_community` is not zero-checked before being used.

```
124         airdropAddr = _airdrop;
```

- `_airdrop` is not zero-checked before being used.

```
125         nextGameAddr = _nextGame;
```

- `_nextGame` is not zero-checked before being used.

```
30          minter = _minter;
```

- `_minter` is not zero-checked before being used.

Recommendation

It is recommended to add a zero-check for the passed-in address value to prevent unexpected errors.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

RCO-04 | POTENTIAL DENIAL OF SERVICE CAUSED BY BUYER

Category	Severity	Location	Status
Denial of Service	● Minor	contracts/RapChain.sol: 324	● Acknowledged

Description

The `distribute()` function checks that the input parameter `len` is exactly one more than the current `chain.len` :

```
323         // Check if the length of the chain is valid
324         require(len == chain.len + 1, "len");
```

The `len` is used in conjunction with a chain ID (`id`) to create a payload that is signed by a signer wallet. This signature is necessary for a buyer to proceed with purchasing a rap chain. However, if a buyer decides not to purchase after receiving the signature, other buyers are blocked from proceeding because they must wait for the current `len` to be consumed, potentially leading to a Denial of Service (DoS) condition.

Recommendation

Please confirm the approach of sharing the purchase signature and avoid DoS.

Alleviation

[RapChain Team, 01/16/2024]:

1. Non-blocking Signature Mechanism: The design of the RapChain protocol's signature mechanism allows for the generation of appropriate signatures using the same len value to overwrite previous ones. This means that even if a buyer chooses not to use their signature, it does not block other buyers from proceeding with their purchases.

2. Competitive Purchase Design: The mechanism's design is intended to foster competition among users to buy at a given len. This competitive aspect is a core feature of the game, encouraging active participation and engagement from buyers.

3. Operational Resilience: The protocol's ability to generate new signatures for the same len value ensures operational continuity and resilience. This design choice mitigates the risk of a single buyer's inaction affecting the overall functionality of the game.

RCO-05 | CHECK FOR "CLAIMENABLE" FLAG WHEN REFERRER WITHDRAWS REWARD

Category	Severity	Location	Status
Logical Issue	● Minor	contracts/RapChain.sol: 282	● Resolved

Description

The `claimEnable` flag intended to control the ability of users to claim rewards. This flag acts as a switch to enable or disable the reward claiming functionality. However, the `claimReferrer()` function does not check the status of `claimEnable` before proceeding with the reward claim process.

Recommendation

Check the `claimEnable` flag in the `claimReferrer()` function.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

RCO-10 | POTENTIAL DIVIDE BY ZERO

Category	Severity	Location	Status
Logical Issue	● Minor	contracts/RapChain.sol: 186	● Resolved

Description

If the game does not have any participants, the `count` could be zero, performing division by zero would raise an error and revert the transaction.

```
186                 chain.pool += sum / count;
```

The expression `sum / count` may divide by zero.

Recommendation

It is recommended to either reformulate the divisor expression, or to use conditionals or require statements to rule out the possibility of a divide-by-zero.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

RCO-12 | THE `RapChain` CONTRACT CAN BE REINITIALIZED

Category	Severity	Location	Status
Logical Issue	● Minor	contracts/RapChain.sol: 122	● Acknowledged

Description

The `initialize()` function is intended to set initial state variables during the contract's deployment. However, this function lacks the necessary modifiers or state checks to prevent it from being executed more than once.

Recommendation

Consider adding a check to ensure the `initialize()` function can only be executed once.

Alleviation

[RapChain Team, 01/16/2024]: The team acknowledged the finding and decided not to change the current codebase.

RCO-13 | POTENTIAL MALFORMED NFT TOKEN ID

Category	Severity	Location	Status
Logical Issue	● Minor	contracts/RapChain.sol: 388~395	● Resolved

Description

The NFT ID is composed of two parts:

- chain ID: `nftId / 100000`
- len: `nftId % 100000`

If the `len` value exceeds 100,000, the chain ID and `len` value parsed from the generated NFT ID are incorrect.

Scenario

If `id = 1` and `len = 100001`, `genNFTId()` returns "200001", indicating that the chain ID is 2 and the `len` is 1.

Recommendation

Consider using a larger denominator, such as e18.

Alleviation

[RapChain Team, 01/19/2024]: The team heeded the advice and resolved the issue in commit 78fbd18644d38ab9d968fd0aa1c2c4d113b10058.

RCO-16 | CHECK-EFFECTS-INTERACTIONS PATTERN VIOLATION

Category	Severity	Location	Status
Concurrency	● Minor	contracts/RapChain.sol: 285, 354	● Resolved

Description

This Checks-Effects-Interactions Pattern is a best practice for writing secure smart contracts that involves performing all state changes before making any external function calls.

```
285         payable(msg.sender).transfer(amount);
286         totalWithdrawal[msg.sender] += amount;
287         referrers[msg.sender] = 0;
```

```
354         IRapNFT(rapNFT).mint(msg.sender, nftId);
355
356         // Set the rewardPerBuy for the NFT
357         debts[nftId] = rewardPerBuy;
```

Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts to prevent unexpected behavior.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

RCO-18 | THE `referrer` COULD BE ANY ADDRESS

Category	Severity	Location	Status
Access Control	● Minor	contracts/RapChain.sol: 335~336	● Resolved

Description

The function for buying a rap chain allows players to specify a `referrer` address. This `referrer` is then awarded a commission of 7% of the buy-in amount. The current implementation does not require the `referrer` to be a pre-existing participant within the system. Additionally, any address can be set as a `referrer`, provided it is not the same as `msg.sender`.

Recommendation

Consider adding restrictions to the referrer address to prevent users from using another wallet as the referrer to pay less.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

RCO-17 | INCONSISTENT COMMENT AND CODE

Category	Severity	Location	Status
Inconsistency	● Informational	contracts/RapChain.sol: 224	● Resolved

Description

The comment 'count how many chains are the longest.' in the `settle()` function does not reflect the code logic.

Recommendation

We recommend checking the current implementation and correcting the inconsistency.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

OPTIMIZATIONS | RAPCHAIN-AUDIT

ID	Title	Category	Severity	Status
RCO-01	Variables That Could Be Declared As Immutable	Gas Optimization	Optimization	● Resolved
RCO-02	Inefficient Memory Parameter	Inconsistency	Optimization	● Resolved
RCO-14	Unused `airdropAddr`	Coding Style, Gas Optimization	Optimization	● Resolved

RCO-01 | VARIABLES THAT COULD BE DECLARED AS IMMUTABLE

Category	Severity	Location	Status
Gas Optimization	● Optimization	contracts/RapChain.sol: 67	● Resolved

Description

The linked variables assigned in the constructor can be declared as `immutable`. Immutable state variables can be assigned during contract creation but will remain constant throughout the lifetime of a deployed contract. A big advantage of immutable variables is that reading them is significantly cheaper than reading from regular state variables since they will not be stored in storage.

Recommendation

We recommend declaring these variables as immutable.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

RCO-02 | INEFFICIENT MEMORY PARAMETER

Category	Severity	Location	Status
Inconsistency	● Optimization	contracts/RapChain.sol: 375	● Resolved

Description

One or more parameters with `memory` data location are never modified in their functions and those functions are never called internally within the contract. Thus, their data location can be changed to `calldata` to avoid the gas consumption copying from calldata to memory.

```
375      function buy(bytes memory signature, uint id, uint len, address referrer)
payable nonReentrant whenNotPaused public {
```

`buy` has memory location parameters: `signature` .

Recommendation

We recommend changing the parameter's data location to `calldata` to save gas.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

RCO-14 | UNUSED `airdropAddr`

Category	Severity	Location	Status
Coding Style, Gas Optimization	● Optimization	contracts/RapChain.sol: 64	● Resolved

Description

The state variable `airdropAddr` is initialized within the `initialize()` function. This variable is also subjected to a non-zero check within the `start()` function. However, it is not utilized anywhere within the contract's functional operations.

Recommendation

If `airdropAddr` is intended to be used in the game logic and contributes to the contract's intended functionality, it is recommended to implement the necessary logic that utilizes this state variable. If `airdropAddr` is a remnant from previous versions of the contract or is not needed for the contract's functionality, it is recommended to remove the variable and any associated checks to reduce gas costs.

Alleviation

[RapChain Team, 01/16/2024]: The team heeded the advice and resolved the issue in commit c93e31b168771d74d9f914f0e6d2f14794856351.

FORMAL VERIFICATION | RAPCHAIN-AUDIT

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied formal verification to prove that important functions in the smart contracts adhere to their expected behaviors.

Considered Functions And Scope

In the following, we provide a description of the properties that have been used in this audit. They are grouped according to the type of contract they apply to.

Verification of Standard ERC-721 Properties

Properties for standard ERC-721 contracts (note that `safeTransferFrom` function is not included in the verification):

Property Name	Title
erc721common-approve-revert-invalid-token	`approve` Fails For Calls with Invalid Tokens
erc721common-approve-revert-not-allowed	`approve` Prevents Unpermitted Approvals
erc721common-approve-set-correct	`approve` Sets Approval
erc721common-approve-succeed-normal	`approve` Returns for Valid Inputs
erc721common-setapprovalforall-set-correct	`setApprovalForAll` Approves Operator
erc721common-isapprovedforall-correct	`isApprovedForAll` Returns Correct Approvals
erc721common-isapprovedforall-succeed	`isApprovedForAll` Always Succeeds
erc721common-isapprovedforall-change-state	`isApprovedForAll` Does Not Change the Contract's State
erc721common-getapproved-revert-zero	`getApproved` Fails on Invalid Tokens
erc721common-getapproved-correct-value	`getApproved` Returns Correct Approved Address
erc721common-getapproved-succeed-normal	`getApproved` Succeeds For Valid Tokens
erc721common-getapproved-change-state	`getApproved` Does Not Change the Contract's State
erc721common-ownerof-revert	`ownerOf` Fails On Invalid Tokens
erc721common-ownerof-correct-owner	`ownerOf` Returns the Correct Owner
erc721common-ownerof-succeed-normal	`ownerOf` Succeeds For Valid Tokens

Property Name	Title
erc721common-ownerof-no-change-state	`ownerOf` Does Not Change the Contract's State
erc721common-balanceof-revert	`balanceOf` Fails on the Zero Address
erc721common-balanceof-correct-count	`balanceOf` Returns the Correct Value
erc721common-balanceof-succeed-normal	`balanceOf` Succeeds on Valid Inputs
erc721common-balanceof-no-change-state	`balanceOf` Does Not Change the Contract's State
erc721common-supportsinterface-correct-erc721	`supportsInterface` Signals Support for `ERC721`
erc165-supportsinterface-correct-false	`supportsInterface` Returns `False` for Id 0xffffffff
erc165-supportsinterface-correct-erc165	`supportsInterface` Signals Support for ERC165
erc165-supportsinterface-succeed-always	`supportsInterface` Always Succeeds
erc721common-transferfrom-revert-exceed-approval	`transferFrom` Fails for Token Transfers without Approval
erc721common-transferfrom-revert-not-owned	`transferFrom` Fails if `From` Is Not Token Owner
erc165-supportsinterface-no-change-state	`supportsInterface` Does Not Change the Contract's State
erc721common-transferfrom-revert-invalid	`transferFrom` Fails for Invalid Tokens
erc721common-transferfrom-correct-state-approval	`transferFrom` Has Expected Approval Changes
erc721common-transferfrom-correct-state-owner	`transferFrom` Has Expected Ownership Changes
erc721common-transferfrom-correct-state-balance	`transferFrom` Keeps Balances Constant Except for From and To
erc721common-transferfrom-revert-zero-argument	`transferFrom` Fails for Transfers with Zero Address Arguments
erc721common-transferfrom-correct-owner-to	`transferFrom` Transfers Ownership
erc721common-transferfrom-correct-approval	`transferFrom` Updates the Approval Correctly
erc721common-transferfrom-correct-increase	`transferFrom` Transfers the Complete Token in Transfers
erc721-transferfrom-succeed-normal	`transferFrom` Succeeds on Valid Inputs
erc721common-transferfrom-correct-balance	`transferFrom` Sum of Balances is Constant

Property Name	Title
erc721common-setapprovalforall-multiple	`setApprovalForAll` Can Set Multiple Operators
erc721common-setapprovalforall-change-state	`setApprovalForAll` Has No Unexpected State Changes
erc721common-setapprovalforall-succeed-normal	`setApprovalForAll` Returns for Valid Inputs
erc721common-approve-change-state	`approve` Has No Unexpected State Changes
erc721common-supportsinterface-metadata	`supportsInterface` Signals that ERC721Metadata is Implemented

Verification Results

In the remainder of this section, we list all contracts where formal verification of at least one property was not successful. There are several reasons why this could happen:

- False: The property is violated by the project.
- Inconclusive: The proof engine cannot prove or disprove the property due to timeouts or exceptions.
- Inapplicable: The property does not apply to the project.

Detailed Results For Contract RapNFT (contracts/RapNFT.sol) In Commit aa0665ef503fdca40ed9669b5cd7b10cc1ce6b90

Verification of Standard ERC-721 Properties

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc721common-approve-revert-invalid-token	● Inconclusive	
erc721common-approve-revert-not-allowed	● Inconclusive	
erc721common-approve-set-correct	● Inconclusive	
erc721common-approve-succeed-normal	● Inconclusive	
erc721common-approve-change-state	● Inconclusive	

Detailed Results for Function `setApprovalForAll`

Property Name	Final Result	Remarks
erc721common-setapprovalforall-set-correct	● True	
erc721common-setapprovalforall-multiple	● True	
erc721common-setapprovalforall-change-state	● Inconclusive	
erc721common-setapprovalforall-succeed-normal	● True	

Detailed Results for Function `isApprovedForAll`

Property Name	Final Result	Remarks
erc721common-isapprovedforall-correct	● True	
erc721common-isapprovedforall-succeed	● True	
erc721common-isapprovedforall-change-state	● True	

Detailed Results for Function `getApproved`

Property Name	Final Result	Remarks
erc721common-getapproved-revert-zero	● Inconclusive	
erc721common-getapproved-correct-value	● Inconclusive	
erc721common-getapproved-succeed-normal	● Inconclusive	
erc721common-getapproved-change-state	● True	

Detailed Results for Function `ownerOf`

Property Name	Final Result	Remarks
erc721common-ownerof-revert	● Inconclusive	
erc721common-ownerof-correct-owner	● Inconclusive	
erc721common-ownerof-succeed-normal	● Inconclusive	
erc721common-ownerof-no-change-state	● True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc721common-balanceof-revert	● True	
erc721common-balanceof-correct-count	● True	
erc721common-balanceof-succeed-normal	● True	
erc721common-balanceof-no-change-state	● True	

Detailed Results for Function `supportsInterface`

Property Name	Final Result	Remarks
erc721common-supportsinterface-correct-erc721	● True	
erc165-supportsinterface-correct-false	● True	
erc165-supportsinterface-correct-erc165	● True	
erc165-supportsinterface-succeed-always	● True	
erc165-supportsinterface-no-change-state	● True	
erc721common-supportsinterface-metadata	● True	

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc721common-transferfrom-revert-exceed-approval	● Inconclusive	
erc721common-transferfrom-revert-not-owned	● Inconclusive	
erc721common-transferfrom-revert-invalid	● Inconclusive	
erc721common-transferfrom-correct-state-approval	● Inconclusive	
erc721common-transferfrom-correct-state-owner	● Inconclusive	
erc721common-transferfrom-correct-state-balance	● True	
erc721common-transferfrom-revert-zero-argument	● Inconclusive	
erc721common-transferfrom-correct-owner-to	● Inconclusive	
erc721common-transferfrom-correct-approval	● Inconclusive	
erc721common-transferfrom-correct-increase	● Inconclusive	
erc721-transferfrom-succeed-normal	● Inconclusive	
erc721common-transferfrom-correct-balance	● True	

APPENDIX | RAPCHAIN-AUDIT

Finding Categories

Categories	Description
Gas Optimization	Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Concurrency	Concurrency findings are about issues that cause unexpected or unsafe interleaving of code executions.
Access Control	Access Control findings are about security vulnerabilities that make protected assets unsafe.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

Details on Formal Verification

Some Solidity smart contracts from this project have been formally verified. Each such contract was compiled into a mathematical model that reflects all its possible behaviors with respect to the property. The model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

The following assumptions and simplifications apply to our model:

- Certain low-level calls and inline assembly are not supported and may lead to a contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for property specifications

All properties are expressed in a behavioral interface specification language that CertiK has developed for Solidity, which allows us to specify the behavior of each function in terms of the contract state and its parameters and return values, as well as contract properties that are maintained by every observable state transition. Observable state transitions occur when the contract's external interface is invoked and the invocation does not revert, and when the contract's Ether balance is changed by the EVM due to another contract's "self-destruct" invocation. The specification language has the usual Boolean connectives, as well as the operator `\old` (used to denote the state of a variable before a state transition), and several types of specification clause:

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates to reason about the validity of atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `requires [cond]` - the condition `cond`, which refers to a function's parameters, return values, and contract state variables, must hold when a function is invoked in order for it to exhibit a specified behavior.
- `ensures [cond]` - the condition `cond`, which refers to a function's parameters, return values, and both `\old` and current contract state variables, is guaranteed to hold when a function returns if the corresponding requires condition held when it was invoked.
- `invariant [cond]` - the condition `cond`, which refers only to contract state variables, is guaranteed to hold at every observable contract state.
- `constraint [cond]` - the condition `cond`, which refers to both `\old` and current contract state variables, is guaranteed to hold at every observable contract state except for the initial state after construction (because there is no previous state); constraints are used to restrict how contract state can change over time.

Description of the Analyzed ERC-721 Properties

Properties related to function `approve`

erc721common-approve-change-state

All calls of the form `approve(to, tokenId)` must only update the allowance mapping according to a valid token `tokenId` and the address `to`, and incur no other state changes.

Specification:

```
assignable getApproved(tokenId);
```

erc721common-approve-revert-invalid-token

All calls of the form `approve(to, tokenId)` must fail for an invalid token.

Specification:

```
reverts_when ownerOf(tokenId) == address(0);
```

erc721common-approve-revert-not-allowed

All calls of the form `approve(to, tokenId)` must fail if the message sender is not permitted to access token `tokenId`.

Specification:

```
reverts_when (ownerOf(tokenId) != msg.sender) &&
!isApprovedForAll(ownerOf(tokenId),msg.sender);
```

erc721common-approve-set-correct

Any returning call of the form `approve(to, tokenId)` must approve the address `to` for token `tokenId`.

Specification:

```
requires ownerOf(tokenId) != address(0);
requires (ownerOf(tokenId) == msg.sender)||
isApprovedForAll(ownerOf(tokenId),msg.sender);
ensures getApproved(tokenId) == to;
```

erc721common-approve-succeed-normal

All calls of the form `approve(to, tokenId)` must return if

- the sender is the owner or an authorized operator of the owner
- the token `tokenId` is valid and
- the execution does not run out of gas.

Specification:

```
requires ownerOf(tokenId) != address(0);
requires ownerOf(tokenId) != to;
requires (ownerOf(tokenId) == msg.sender) ||
isApprovedForAll(ownerOf(tokenId),msg.sender);
ensures true;
reverts_only_when false;
```

Properties related to function `setApprovalForAll`

erc721common-setapprovalforall-change-state

All calls of the form `setApprovalForAll(operator, approved)` must only update the approval mapping according to the message sender, the address `operator` and the Boolean value `approved` but incur no other state changes.

Specification:

```
assignable isApprovedForAll(msg.sender,operator);
```

erc721common-setapprovalforall-multiple

Calls of the form `setApprovalForAll(operator, approved)` must be able to set multiple operators for the tokens of the message sender.

Specification:

```
requires approved;
ensures isApprovedForAll(msg.sender,operator);
ensures (\forall address op1 |(op1 != address(0)) &&
\old(isApprovedForAll(msg.sender,op1)) :: isApprovedForAll(msg.sender,op1));
```

erc721common-setapprovalforall-set-correct

All non-reverting calls of the form `setApprovalForAll(operator, approved)` must set the approval of a non-zero address `operator` according to the Boolean value `approved` .

Specification:

```
requires operator != address(0);
ensures isApprovedForAll(msg.sender,operator) == approved;
```

erc721common-setapprovalforall-succeed-normal

Calls of the form `setApprovalForAll(operator, approved)` must return if

- the message sender is not the `operator` ,
- `operator` is not the zero address and
- the execution does not run out of gas.

Specification:

```
requires msg.sender != operator;
requires operator != address(0);
ensures true;
reverts_only_when false;
```

Properties related to function `isApprovedForAll`

erc721common-isapprovedforall-change-state

Function `isApprovedForAll` does not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc721common-isapprovedforall-correct

Invocations of `isApprovedForAll(owner, operator)` must return whether a non-zero address `operator` is approved for tokens of a non-zero address `owner` , or return false.

Specification:

```
requires owner != address(0);
requires operator != address(0);
ensures \result == isApprovedForAll(owner,operator);
```

erc721common-isapprovedforall-succeed

Function `isApprovedForAll` does always succeed, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `getApproved`

erc721common-getapproved-change-state

Function `getApproved` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc721common-getapproved-correct-value

Invocations of `getApproved(token)` must return the approved address of a valid `token` .

Specification:

```
ensures (\result == \old(getApproved(tokenId))) || (\result == address(0));
```

erc721common-getapproved-revert-zero

Invocations of `getApproved(token)` with an invalid token must fail.

Specification:

```
reverts_when ownerOf(tokenId) == address(0);
```

erc721common-getapproved-succeed-normal

Function `getApproved` must always succeed for valid tokens, assuming that its execution does not run out of gas.

Specification:

```
requires ownerOf(tokenId) != address(0);
ensures true;
reverts_only_when false;
```

Properties related to function `ownerOf`

erc721common-ownerof-correct-owner

Invocations of `ownerOf(token)` must return the owner for a valid token `token` that is held in the contract's owner mapping.

Specification:

```
requires ownerOf(tokenId) != address(0);
ensures \result == \old(ownerOf(tokenId));
```

erc721common-ownerof-no-change-state

Function `ownerOf` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc721common-ownerof-revert

Invocations of `ownerOf(token)` must fail for an invalid token.

Specification:

```
reverts_when ownerOf(tokenId) == address(0);
```

erc721common-ownerof-succeed-normal

Function `ownerOf(token)` must always succeed for valid tokens if it does not run out of gas.

Specification:

```
requires ownerOf(tokenId) != address(0);
ensures true;
reverts_only_when false;
```

Properties related to function `balanceOf`

erc721common-balanceof-correct-count

Invocations of `balanceOf(owner)` must return the value that is held in the balance mapping for address `owner`.

Specification:

```
ensures \result == \old(balanceOf(owner));
```

erc721common-balanceof-no-change-state

Function `balanceOf` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc721common-balanceof-revert

Invocations of `balanceOf(owner)` must fail if the address `owner` is the zero address.

Specification:

```
reverts_when owner == address(0);
```

erc721common-balanceof-succeed-normal

All invocations of `balanceOf(owner)` must succeed if the address `owner` is not zero and it does not run out of gas.

Specification:

```
requires owner != address(0);
ensures true;
reverts_only_when false;
```

Properties related to function `supportsInterface`

erc165-supportsinterface-correct-erc165

Invocations of `supportsInterface(id)` must signal that the interface `ERC165` is implemented.

Specification:

```
requires interfaceId == 0x01ffc9a7;
ensures \result;
```

erc165-supportsinterface-correct-false

Invocations of `supportsInterface(id)` with `id` 0xffffffff must return `false`.

Specification:

```
requires interfaceId == 0xffffffff;
ensures !\result;
```

erc165-supportsinterface-no-change-state

Function `supportsInterface` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc165-supportsinterface-succeed-always

Function `supportsInterface` must always succeed if it does not run out of gas.

Specification:

```
reverts_only_when false;
```

erc721common-supportsinterface-correct-erc721

Invocations of `supportsInterface(id)` must signal that the interface `ERC721` is implemented.

Specification:

```
requires interfaceId == 0x80ac58cd;
ensures esult;
```

erc721common-supportsinterface-metadata

A call of `supportsInterface(interfaceId)` with the interface id of ERC721Metadata must return true.

Specification:

```
requires interfaceId == 0x5b5e139f;
ensures \result;
```

Properties related to function `transferFrom`

erc721-transferfrom-succeed-normal

All invocations of `transferFrom(from, to, tokenId)` must succeed if

- address `from` is the owner of token `tokenId` , *it is not a self transfer,
- the sender is approved to transfer token `tokenId` ,
- transferring the token to the address `to` does not lead to an overflow of the recipient's balance, and
- the supplied gas suffices to complete the call.

Specification:

```
requires from != address(0);
requires to != address(0);
requires from != to;
requires from == ownerOf(tokenId);
requires balanceOf(from) > 0;
requires balanceOf(to) < type(uint256).max;
requires (msg.sender == from)||(getApproved(tokenId) == msg.sender) ||
isApprovedForAll(from,msg.sender);
ensures true;
reverts_only_when false;
```

erc721common-transferfrom-correct-approval

All non-reverting invocations of `transferFrom(from, to, tokenId)` that return must remove any approval for token `tokenId` .

Specification:

```
ensures getApproved(tokenId) == address(0);
```

erc721common-transferfrom-correct-balance

All non-reverting invocations of `transferFrom(from, to, tokenId)` must keep the sum of token balances constant.

Specification:

```
requires from != address(0);
requires to != address(0);
requires balanceOf(from) > 0;
requires balanceOf(to) < type(uint256).max;
ensures (\old(balanceOf(from)) - balanceOf(from)) == (balanceOf(to) -
\old(balanceOf(to)));
```

erc721common-transferfrom-correct-increase

All invocations of `transferFrom(from, to, tokenId)` that succeed must subtract a token from the balance of address `from` and add the token to the balance of address `to`.

Specification:

```
requires from != to;
requires balanceOf(from) > 0;
requires balanceOf(to) < type(uint256).max;
ensures (balanceOf(from) == \old(balanceOf(from)) - 1) && (balanceOf(to) ==
\old(balanceOf(to)) + 1);
  also
requires from == to;
requires ownerOf(tokenId) == from;
ensures balanceOf(from) == \old(balanceOf(from));
```

erc721common-transferfrom-correct-owner-to

All non-reverting invocations of `transferFrom(from, to, tokenId)` must transfer the ownership of token `tokenId` to the address `to`.

Specification:

```
requires from != address(0);
requires to != address(0);
requires (msg.sender == from) || (getApproved(tokenId) == msg.sender) ||
isApprovedForAll(from,msg.sender);
ensures ownerOf(tokenId) == to;
```

erc721common-transferfrom-correct-state-approval

All non-reverting invocations of `transferFrom(from, to, tokenId)` must remove only approvals for token `tokenId`

Specification:

```
ensures (\forall uint id | id!=tokenId :: \old(getApproved(id))==getApproved(id));
```

erc721common-transferfrom-correct-state-balance

All non-reverting invocations of `transferFrom(from, to, tokenId)` must only modify the balance of the addresses `from` and `to`.

Specification:

```
ensures (\forall address adr | (adr!=from) && (adr!=to) :: \old(balanceOf(adr))==
balanceOf(adr));
```

erc721common-transferfrom-correct-state-owner

All non-reverting invocations of `transferFrom(from, to, tokenId)` must only modify the ownership of token `tokenId`.

Specification:

```
ensures (\forall uint id | id!=tokenId :: \old(ownerOf(id))==ownerOf(id));
```

erc721common-transferfrom-revert-exceed-approval

Any call of the form `transferFrom(from, to, tokenId)` must fail if the sender is neither the token owner nor an operator of the token owner nor approved for token `tokenId`.

Specification:

```
reverts_when (msg.sender != from) && (getApproved(tokenId) != msg.sender) &&
!isApprovedForAll(from,msg.sender);
```

erc721common-transferfrom-revert-invalid

All calls of the form `transferFrom(from, to, tokenId)` must fail for any invalid token.

Specification:

```
reverts_when ownerOf(tokenId) == address(0);
```

erc721common-transferfrom-revert-not-owned

Any call of the form `transferFrom(from, to, tokenId)` must fail if address 'from' is not the owner of token `tokenId`.

Specification:

```
reverts_when ownerOf(tokenId) != from;
```

erc721common-transferfrom-revert-zero-argument

All calls of the form `transferFrom(from, to, tokenId)` must fail for transfers from or to the zero address.

Specification:

```
reverts_when to == address(0);
also
reverts_when from == address(0);
```

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK’s prior written consent in each instance.

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER’S OR ANY OTHER PERSON’S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.