oncha

Preliminary Comments

CertiK Assessed on Dec 9th, 2025

CertiK Assessed on Dec 9th, 2025

oncha

These preliminary comments were prepared by CertiK.

Executive Summary

TYPES	ECOSYSTEM	METHODS
Oracle	Binance Smart Chain (BSC)	Manual Review, Static Analysis

LANGUAGE	TIMELINE
Solidity	Preliminary comments published on 11/27/2025

Vulnerability Summary

17	7	1	8	0	1
Total Findings	Resolved	Partially Resolved	Acknowledged	Declined	Pending

1	Centralization	1 Acknowledged	Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.
0	Critical		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.
0	Major		Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
1	Medium	1 Partially Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
7	Minor	5 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.
7	Informational	2 Resolved, 5 Acknowledged	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.
1	Discussion	1 Pending	The impact of the issue is yet to be determined, hence requires further clarifications from the project team.

TABLE OF CONTENTS | ONCHA

Disclaimer

CODEBASE | ONCHA

Repository

- NFT
- RandomNumber

Commit

- 0x38a3759e4252bb9caafe255a594edf6045c84a99
- 0xb8dee596efa0f59cc8434088fffb28cdd8e22d3e
- 0x5b2f1e8a6ba28b40acedc49af5ea66a91501325d
- 0x6643adcdb60ff832a7e0ec823572edffb6c08f5b
- 0xe0b676819a5c371670aaf2a231566c065e91bed1

AUDIT SCOPE | ONCHA

mainnet
contracts/NFT.sol
contracts/RandomNumber.sol
contracts/RandomSeed.sol
contracts/RandomSeed.sol
contracts/RandomNumber.sol
contracts/NFT.sol
contracts/RandomNumber.sol
contracts/RandomSeed.sol

APPROACH & METHODS | ONCHA

This audit was conducted for oncha to evaluate the security and correctness of the smart contracts associated with the oncha project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Manual Review and Static Analysis.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

REVIEW NOTES | ONCHA

Overview

The **oncha** project coordinates a series of smart contracts aimed at supporting NFT purchase and generating random number on the blockchain.

External Dependencies

In **oncha**, the module inherits or uses a few of the depending injection contracts or addresses to fulfill the need of its business logic. The scope of the audit treats third party entities as black boxes and assume their functional correctness. However, in the real world, third parties can be compromised and this may lead to lost or stolen assets.

Addresses

The following addresses interact at some point with specified contracts, making them an external dependency. All of the following values are initialized either at deployment time or by specific functions in smart contracts.

NFT

- `paymentToken` .

RandomNumber

- `s_vrfCoordinator` , `s_owner` .

We assume these contracts or addresses are valid and non-vulnerable actors and implementing proper logic to collaborate with the current project.

Also, the following library/contract are considered as the third-party dependencies:

- @openzeppelin/contracts/
- @chainlink/contracts/

Privileged Functions

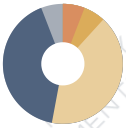
In the **oncha** project, the privileged roles are adopted to ensure the dynamic runtime updates of the project, which are specified in the `Centralization` finding.

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to best serve the community. It is also worth noting the potential drawbacks of these functions, which should be clearly stated through the client's action/plan. Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of the

`Timelock` contract.

FINDINGS | ONCHA



17	0	1	0	1	7	7	1
Total Findings	Critical	Centralization	Major	Medium	Minor	Informational	Discussion

This report has been prepared for oncha to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 17 issues were identified. Leveraging a combination of Manual Review & Static Analysis the following findings were uncovered:

ID	Title	Category	Severity	Status
ONC-04	**Centralization Related Risks**	**Centralization**	**Centralization**	**Acknowledged**
ONC-15	Missing Access Control And Time Constraints In `finalizeCurrentRound`	Design Issue	Medium	Partially Resolved
ONC-05	Missing Input Validation In `getRoundResultExtended()` Function	Incorrect Calculation	Minor	Resolved
ONC-06	Missing Emit Events	Inconsistency	Minor	Resolved
ONC-07	Missing Duplicate-Recipient Validation In `distributeRewards` May Lead To Double Payments	Volatile Code	Minor	Acknowledged
ONC-08	Overwriting Of `s_pendingRound` Causes Lost Or Invalid VRF Requests	Volatile Code	Minor	Acknowledged
ONC-09	Calling `NFT._verifyAndProcess` Directly Bypasses Contract Paused Mechanism	Logical Issue	Minor	Resolved
ONC-10	Potential Overflow And Out-Of-Gas Vulnerabilities In `getRoundResultExtended`	Denial of Service	Minor	Resolved
ONC-16	Sampling Off-By-One Error Introduces Deterministic Bias Toward Minimum Value	Logical Issue	Minor	Resolved
ONC-01	Signature Replay Attack	Logical Issue	Informational	Acknowledged

ID	Title	Category	Severity	Status
ONC-02	Missing `nftNumber` In Signed Message	Design Issue	Informational	Acknowledged
ONC-03	Unclear Integration Between `NFT` And `RandomNumber` Contracts	Design Issue	Informational	Acknowledged
ONC-11	`fulfillRandomWords()` May Revert	Volatile Code	Informational	Acknowledged
ONC-12	Potential Incorrect Function Visibility Or Naming Conventioin Violation	Code Optimization	Informational	Resolved
ONC-13	Debug And Test Logic In `RandomNumber` Contract Increases Complexity And Reduces Maintainability	Coding Style	Informational	Acknowledged
ONC-14	Unnecessary Fund VRF Consumer Contract	Design Issue	Informational	Resolved
ONC-17	Potential Predictable Outputs In `getRoundResultExtended` Due To Seed Reuse Across Multiple Blocks/Transactions	Design Issue	Discussion	Pending

ONC-04 | Centralization Related Risks

Category	Severity	Location	Status
Centralization	Centralization		Acknowledged

Description

In the contract `NFT` , the role `DEFAULT_ADMIN_ROLE` has authority over the following functions:

- `setNftPrice()`
- `addOperator()`
- `removeOperator()`
- `addManager()`
- `removeManager()`
- `addDistributor()`
- `removeDistributor()`
- `addPermitter()`
- `removePermitter()`
- `rescueTokens()`
- `pause()`
- `unpause()`
- `grantRole()`
- `revokeRole()`

Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow an attacker to arbitrarily set sale prices, reassign or strip every operational role, seize arbitrary ERC20/Ether from the contract, and globally pause or resume trading, effectively giving full control over protocol configuration and treasury. Because `DEFAULT_ADMIN_ROLE` is also the admin for every subordinate role in OpenZeppelin's `AccessControl` , it can grant or revoke any role at will.

In the contract `NFT` , the role `OPERATOR_ROLE` has authority over the following functions:

- `_verifyAndProcess()`
- `buyBatch()`

Any compromise to the `OPERATOR_ROLE` account may allow an attacker to validate arbitrary purchase payloads and execute large batch purchases regardless of business logic expectations, letting them front-run users or drain user allowances while the contract is paused elsewhere.

In the contract `NFT` , the role `MANAGER_ROLE` has authority over the following functions:

- `updateCheckPoint1()`
- `updateCheckPoint2()`

- `withdrawTokenByAmount()`
- `withdrawByPercent()`

Any compromise to the `MANAGER_ROLE` account may allow an attacker to modify internal accounting checkpoints and withdraw arbitrary ERC20 balances (by fixed amount or percentage), effectively granting treasury-drain capabilities.

In the contract `NFT`, the role `DISTRIBUTOR_ROLE` has authority over the following functions:

- `distributeRewards()`

Any compromise to the `DISTRIBUTOR_ROLE` account may allow unauthorized dispersal of the contract's token reserves to attacker-controlled recipients.

In the contract `NFT`, the role `PERMIT_ROLE` has authority over the following functions:

- `approveWithPermit()`
- `batchApproveWithPermit()`

Any compromise to the `PERMIT_ROLE` account may allow an attacker to submit arbitrary user EIP-2612 permits, thereby setting allowances that let the NFT contract pull funds from victims without their intention.

In the contract `RandomSeed`, the role `owner` has authority over the following functions:

- `updateChainLinkConfig()`
- `updateCallbackGasLimit()`
- `togglePaymentMethod()`
- `setBlockWait()`
- `withdrawETH()`
- `transferOwnership()`
- `setCoordinator()`
- `finalizeCurrentRound()`
- `finalizeRound()`

Any compromise to the `owner` account may allow an attacker to reconfigure VRF subscription parameters, change gas budgets, flip between LINK/native payment modes, move any ETH held by the contract, and modify block-wait rules that guard reveal timing, effectively granting total control over randomness generation costs, timing, and treasury.

In the contract `RandomSeed`, the role `s_vrfCoordinator` has authority over the following functions:

- `setCoordinator()`

Any compromise to the `s_vrfCoordinator` account may allow an attacker to update the `s_vrfCoordinator`.

In the contract `RandomNumber`, the role `owner` has authority over the following functions:

- `generateNextRoundSeed()`
- `advanceRound()`

- `advanceRoundBy()`

Any compromise to the `owner` account may allow an attacker to unilaterally advance rounds, submit arbitrary commitments, and request new VRF seeds at will, enabling them to manipulate or halt the randomness lifecycle for all downstream consumers.

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

[oncha, 12/02/2025]:

We acknowledge this risk and have plans already to reduce its impact, and we think that is good enough.

We put the related private keys in a trusted environment, and the private keys are also encrypted at rest. We are also considering putting the owner of the contract into a hardware wallet or in a multi-sig wallet.

We also have a script to closely track the event of these role or ownership changes on the NFT and RandomNumber contract.

We will also document the related accounts and publish them to the client.

[CertiK, 12/02/2025]:

The finding can be updated after relevant information is provided. It is suggested to implement the aforementioned methods to avoid centralized failure. Also, CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

ONC-15 | Missing Access Control And Time Constraints In `finalizeCurrentRound`

Category	Severity	Location	Status
Design Issue	Medium	contracts/RandomSeed.sol (12/01-RandomNumber): 156	Partially Resolved

Description

The `finalizeCurrentRound` function in the `RandomSeed` contract lacks both access control modifiers and time-based constraints. Unlike `generateNextRoundSeed` which requires `onlyOwner`, `finalizeCurrentRound` can be called by any address that possesses the correct `secret`. Additionally, there is no deadline mechanism, allowing the owner to indefinitely delay finalization after seeing the VRF result. This creates two critical vulnerabilities: (1) any party with access to the `secret` can finalize rounds, potentially causing operational interference, and (2) the owner can selectively reveal only favorable outcomes or cause permanent DoS by never finalizing unfavorable rounds, breaking downstream functionality that depends on finalized random numbers.

```
// For useOracle == true
function finalizeCurrentRound(bytes32 secret) external {
    require(useOracle, "Not in oracle mode");
    uint256 round = s_pendingRound.round;
    require(round > 0, "No round pending");
    require(s_roundResults[round] == 0, "Already revealed");
    require(s_pendingRound.vrfResult != 0, "VRF result not yet available");
    require(s_roundFulfilledBlock[round] != 0, "Round not fulfilled");
    bytes32 computedCommitment = keccak256(abi.encodePacked(secret));
    require(computedCommitment == s_pendingRound.commitment, "Invalid secret");

    uint256 finalRandom = uint256(keccak256(abi.encodePacked(
        s_pendingRound.vrfResult,
        secret
    )));

    s_roundResults[round] = finalRandom;
    delete s_pendingRound;
}
```

RandomSeed.sol:138-153

```
function fulfillRandomWords(
    uint256 requestId,
    uint256[] calldata randomWords
) internal override {
    require(useOracle, "fulfillRandomWords should only be called in oracle mode");
    uint256 round = s_requestToRound[requestId];
    require(round == s_pendingRound.round, "Round mismatch");

    uint256 l2BlockNumber = ArbSys(address(100)).arbBlockNumber();
    s_roundFulfilledBlock[round] = l2BlockNumber;
    if (needFinalize) {
        s_pendingRound.vrfResult = randomWords[0];  // VRF result stored on-chain,
publicly visible
    } else {
        s_roundResults[round] = randomWords[0];
    }
}
```

RandomNumber.sol:21-25

```
function generateNextRoundSeed(bytes32 secretCommitment) external onlyOwner returns
(uint256) {
    advanceRound();
    currentRequestId = requestRandomWords(currentRound, secretCommitment);
    return currentRound;
}
```

The `finalizeCurrentRound` function implements a commit-reveal scheme where the owner commits to a `secret` via `keccak256(secret)` before requesting Chainlink VRF, then later reveals the `secret` to compute the final random number. However, the implementation has two critical flaws:

1. **Missing Access Control**: The function has no `onlyOwner` modifier, allowing any address to call it if they possess the `secret`. While the function validates that `keccak256(secret) == commitment`, it doesn't verify that the caller is the original commitment submitter. Operational interference is possible if multiple parties have access to the `secret`.

2. **Missing Time Constraints**: Unlike the local mode's `finalizeRound` which enforces `blockWait` and a 255-block window, the oracle mode has no time-based requirements. The owner can wait indefinitely after the VRF result is stored on-chain.

Recommendation

It's recommended to add access control and time constraints in the round finalization functions.

Alleviation

[oncha, 12/07/2025]:

The issue has been fixed. Here is the new random number contract address:0xe0b676819a5C371670AAf2a231566c065e91beD1

[CertiK, 12/08/2025]:

The team added the `onlyOwner` modifier to both functions without any time constraints. We recommend that the owner executes both functions in a timely manner.

ONC-05 | Missing Input Validation In `getRoundResultExtended()` Function

Category	Severity	Location	Status
Incorrect Calculation	Minor	contracts/RandomNumber.sol (RandomNumber): 66, 79	Resolved

Description

Since solidity version 0.8.0, arithmetic operations revert on underflow and overflow. In the current implementation, the `max - min` will cause an underflow if `min` > `max` that will cause the function to revert.

Recommendation

We recommend ensuring that the `min` is lower than the `max`.

Alleviation

[oncha, 12/02/2025]:
The team heeded the advice and resolved the issue in the updated version 0x6643adcdb60ff832a7e0ec823572edffb6c08f5b.

ONC-06 | Missing Emit Events

Category	Severity	Location	Status
Inconsistency	Minor	contracts/RandomSeed.sol (RandomNumber): 69~79, 226~228	Resolved

Description

The smart contract contains one or more state changes related to arithmetic operations that do not emit events to communicate the changes outside the blockchain, which can lead to difficulties in tracking or verifying these changes and may affect the contract's transparency and auditability.

```
227          callbackGasLimit = newGasLimit;
```

```
75          s_subscriptionId = _subscriptionId;
```

```
77          requestConfirmations = _requestConfirmations;
```

Recommendation

It is suggested to declare and emit corresponding events for all the essential state variables that are possible to be changed during runtime.

Alleviation

[oncha, 12/02/2025]:
The team heeded the advice and resolved the issue in the updated version 0x6643adcdb60ff832a7e0ec823572edffb6c08f5b.

ONC-07 | Missing Duplicate-Recipient Validation In `distributeRewards` May Lead To Double Payments

Category	Severity	Location	Status
Volatile Code	Minor	contracts/NFT.sol (NFT): 291~294	Acknowledged

Description

The `distributeRewards` function allows an authorized distributor to transfer rewards to multiple recipients in a batch:

```
function distributeRewards(
    address[] calldata recipients,
    uint256[] calldata amounts
) external onlyRole(DISTRIBUTOR_ROLE) {
    require(recipients.length == amounts.length, "Length mismatch");

    uint256 length = recipients.length;
    for (uint i = 0; i < length; ) {
        paymentToken.safeTransfer(recipients[i], amounts[i]);
        unchecked {
            ++i;
        }
    }
}
```

However, the function does **not** validate whether the `recipients` array contains duplicate addresses. If duplicate entries exist (whether accidentally or maliciously), the same recipient may receive rewards multiple times within a single execution.

Recommendation

It is suggested to add a check to ensure that the `recipients` list contains no duplicate addresses before processing transfers.

Alleviation

[oncha, 12/02/2025]:
The team acknowledged the issue and stated that we are actually allowing duplicate recipients when paying the rewards.

ONC-08 | Overwriting Of `s_pendingRound` Causes Lost Or Invalid VRF Requests

Category	Severity	Location	Status
Volatile Code	Minor	contracts/RandomSeed.sol (RandomNumber): 97~102	Acknowledged

Description

In the `requestRandomWords` function, the contract assigns the new round request to a single global storage variable `s_pendingRound`:

```
s_pendingRound = PendingRound({
    round: round,
    commitment: secretCommitment,
    vrfResult: 0,
    commitBlock: 0
});
```

However, the contract does **not** verify whether a pending round already exists before overwriting it. If the owner calls `generateNextRoundSeed` (which internally calls `requestRandomWords`) multiple times before a VRF response is fulfilled, each invocation overwrites the previous `s_pendingRound`.

This becomes problematic because the VRF fulfillment logic relies on a strict one-to-one mapping between a request ID and the single `s_pendingRound`:

```
uint256 round = s_requestToRound[requestId];
require(round == s_pendingRound.round, "Round mismatch");
```

As a result, previous `s_pendingRound` data is discarded, making earlier VRF requests impossible to process.

Recommendation

It is suggested to implement proper tracking of pending rounds to avoid overwriting VRF request data.

Alleviation

[oncha, 12/02/2025]:

The team acknowledged the issue and stated that the intention is that if the VRF failed to fulfill within a certain given amount, we are going to overwrite it and consider that unfilled round invalid.

ONC-09 | Calling `NFT._verifyAndProcess` Directly Bypasses Contract Paused Mechanism

Category	Severity	Location	Status
Logical Issue	Minor	contracts/NFT.sol (NFT): 219, 257	Resolved

Description

In the `_verifyAndProcess` function of the `NFT` contract, the function is declared as `external` and does not include the `whenNotPaused` modifier present in `buyBatch`. As a result, any account with the `OPERATOR_ROLE` can directly call `_verifyAndProcess`, bypassing `buyBatch`, the pausing mechanism, and reentrancy protection.

This allows purchases to proceed even when the contract is paused, which may conflict with the expected behavior of the contract.

Additionally, the current `_buySingle` function calls `_verifyAndProcess` via an external call, requiring the NFT contract itself to have the `OPERATOR_ROLE`. If the contract does not hold this role, any purchase attempt will result in an "AccessControlUnauthorizedAccount" error, preventing successful transactions.

```
    function _buySingle(
        BuyRequest calldata request,
        bytes calldata signature
    ) internal returns (bool) {
    @>  try this._verifyAndProcess(request, signature) {
            return true;
        } catch Error(string memory reason) {
            // This catches require() failures like "Expired" and "Invalid
signature"
            emit ProcessingError(
                request.user,
                request.nftType,
                request.nftPage,
                reason
            );
            return false;
        } catch {
```

According to the on-chain contract method hasRole, the current contract has not been granted the `OPERATOR_ROLE`. Therefore, `OPERATOR_ROLE` accounts currently are unable to make purchases through the `buyBatch()` function.

Recommendation

It's recommended to restrict `_verifyAndProcess` to internal usage (e.g., change it to `internal` and call it only from

guarded functions) and update `_buySingle` to remove external call usage.

Alleviation

[oncha, 12/02/2025]:
Fixed it by adding this to ensure internal call only.

So keep it external, so try-catch works.

The idea is that any `_verifyAndProcess` failure should not fail the buyBatch.

[CertiK, 12/02/2025]:
The team resolved the issue by adding `msg.sender` is `address(this)` and changes were reflected in the updated version 0x5b2f1e8a6ba28b40acedc49af5ea66a91501325d.

ONC-10 | Potential Overflow And Out-Of-Gas Vulnerabilities In `getRoundResultExtended`

Category	Severity	Location	Status
Denial of Service	Minor	contracts/RandomNumber.sol (RandomNumber): 63	Resolved

Description

In the contract `RandomNumber`, the function `getRoundResultExtended` attempts to generate `count` unique random numbers in the range `[min, max]` using rejection sampling. For each index `i` it:

- Iterates a `while` loop up to 10,000 attempts, and
- Inside each attempt, runs a `for` loop over all previously generated values:

```
        for (uint8 j = 0; j < i; j++) {
            if (results[j] == newNumber) {
                isUnique = false;
                break;
            }
        }
```

Overflow issue (primary risk)

The inner loop index `j` is declared as `uint8`. When `count > 255`, the outer loop `i` will eventually reach `256`, and the inner loop tries to run with `j` taking values up to `255`. At `j == 255`, the increment `j++` overflows and wraps to `0`. Since Solidity 0.8+ uses checked arithmetic, this overflow produces a `panic: arithmetic underflow or overflow (0x11)` and causes the entire call to revert. As a result, any call with sufficiently large `count` (e.g., >255) will deterministically revert, even if gas is ample, and any on-chain logic that relies on this helper can be permanently DoS'd for those inputs.

Gas complexity issue (secondary risk)

Even after fixing the integer type, the algorithm's worst-case complexity is high:

- Outer loop: `O(count)`
- Inner while loop: up to 10,000 attempts per element (worst-case)
- Uniqueness check: `O(i)` per attempt

When `count` is large and the range `[min, max]` is tight (i.e., `max - min + 1` is close to `count`), many candidates will be rejected as duplicates, driving the loop toward `O(count² * attempts)` work. If this function is ever used in a state-changing context or via off-chain infra with fixed gas limits, an attacker or misconfigured caller can choose extreme parameters (large `count`, tight range) to push the call to genuine out-of-gas, effectively creating a denial-of-service condition for any flow that depends on this helper.

Proof of Concept

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.30;

import "forge-std/Test.sol";
import {RandomNumber} from "../contracts/RandomNumber.sol";

contract RandomSeedForkTest is Test {
    uint256 private constant FORK_BLOCK = 404273288;
    uint256 private constant BLOCK_TIME = 1732665600;
    address private constant RANDOM_NUMBER =
        0xb8dee596eFa0F59cc8434088fFfB28Cdd8E22d3E;

    RandomNumber internal rng;
    address internal owner;

    function setUp() public {
        vm.createSelectFork("arbitrum", FORK_BLOCK);
        vm.warp(BLOCK_TIME);
        rng = RandomNumber(RANDOM_NUMBER);
        owner = rng.owner();

        vm.label(RANDOM_NUMBER, "RandomNumberContract");
        vm.label(owner, "RandomNumberOwner");
    }

    function test_poc1_GetRoundResultExtendedOutOfGas() public {
        vm.prank(owner);
        rng.advanceRound();
        uint256 round = rng.currentRound();
        uint32[] memory results = rng.getRoundResultExtended(
            "test",
            round - 1,
            100,
            100000,
            257,
            0
        );
        for (uint256 i = 0; i < results.length; i++) {
            console.logUint(results[i]);
        }
    }
}
```

Test results:

```
% forge test --mt test_poc1 -vvv
[⠒] Compiling...
[⠒] Compiling 1 files with Solc 0.8.30
[⠒] Solc 0.8.30 finished in 1.25s
Compiler run successful!

Ran 1 test for test/RandomSeed.t.sol:RandomSeedForkTest
[FAIL: panic: arithmetic underflow or overflow (0x11)]
test_poc1_GetRoundResultExtendedOutOfGas() (gas: 7930231)
Traces:
  [7930231] RandomSeedForkTest::test_poc1_GetRoundResultExtendedOutOfGas()
    ├─ [0] VM::prank(RandomNumberOwner:
[0x1801C13F476d16D4D6C5Db1fd92f7Ca5242d84C1])
    │   └─ ← [Return]
    ├─ [8563] RandomNumberContract::advanceRound()
    │   ├─ emit RoundSet(newRound: 118)
    │   └─ ← [Stop]
    ├─ [375] RandomNumberContract::currentRound() [staticcall]
    │   └─ ← [Return] 118
    ├─ [7909830] RandomNumberContract::getRoundResultExtended("test", 117, 100,
100000 [1e5], 257, 0) [staticcall]
    │   └─ ← [Revert] panic: arithmetic underflow or overflow (0x11)
    └─ ← [Revert] panic: arithmetic underflow or overflow (0x11)

Backtrace:
  at RandomNumberContract.getRoundResultExtended
  at RandomSeedForkTest.test_poc1_GetRoundResultExtendedOutOfGas

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 1.15s (22.92ms CPU
time)

Ran 1 test suite in 1.16s (1.15s CPU time): 0 tests passed, 1 failed, 0 skipped (1
total tests)

Failing tests:
Encountered 1 failing test in test/RandomSeed.t.sol:RandomSeedForkTest
[FAIL: panic: arithmetic underflow or overflow (0x11)]
test_poc1_GetRoundResultExtendedOutOfGas() (gas: 7930231)

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

Recommendation

It's recommended to revise code to restrict the `counter` with proper upper limit.

Alleviation

[oncha, 12/02/2025]:

The team heeded the advice and resolved the issue in the updated version

0x6643adcdb60ff832a7e0ec823572edffb6c08f5b.

ONC-16 | Sampling Off-By-One Error Introduces Deterministic Bias Toward Minimum Value

Category	Severity	Location	Status
Logical Issue	Minor	contracts/RandomNumber.sol (12/01-RandomNumber): 86	Resolved

Description

In the `getRoundResultExtended` function of the `RandomNumber` contract, the rejection sampling implementation contains an off-by-one error that introduces a deterministic bias toward the minimum value (`min`) of the requested range.

```
function getRoundResultExtended(string memory name, uint256 round, uint32 min,
uint32 max, uint16 count, uint16 offset)
    external view returns (uint32[] memory) {
    ...
    uint32 range = max - min + 1;
    uint32 maxValid = type(uint32).max - (type(uint32).max % range);

    uint32 candidate=uint32(temp);
    //rejection sampling to eliminate modulo bias
    if(candidate > maxValid){
        continue;
    }

    newNumber = candidate % range + min;
    ...
}
```

The function attempts to eliminate modulo bias by rejecting candidates that fall outside a uniform distribution. The code computes `maxValid = type(uint32).max - (type(uint32).max % range)` and then accepts candidates where `candidate <= maxValid` (i.e., rejects only when `candidate > maxValid`). This implementation flaw causes `candidate == maxValid` to be accepted, which is problematic because:

1. **Mathematical Analysis**: The accepted candidate space is `[0, maxValid]` inclusive, containing `maxValid + 1` values. Since `maxValid` is the largest multiple of `range` that fits within `type(uint32).max` , the total number of accepted values is not evenly divisible by `range` . Specifically, `maxValid % range == 0` , meaning `candidate == maxValid` maps to residue class 0 (i.e., `min`).

2. **Concrete Example**: For a coin flip scenario with `range = 2` (min=0, max=1):

   - `type(uint32).max = 4294967295`
   - `maxValid = 4294967294` (largest even number ≤ 2^32-1)
   - Accepted candidates: `[0, 4294967294]` = 4294967295 values

- Values mapping to `min` (0): All even numbers = 2147483648 occurrences
- Values mapping to `min+1` (1): All odd numbers = 2147483647 occurrences
- **Bias**: `min` appears exactly one more time than `min+1`, creating a `1/4294967295` probability skew

3. **Root Cause**: The rejection condition `if(candidate > maxValid)` should be `if(candidate >= maxValid)` to properly exclude the boundary value. Alternatively, `maxValid` should be computed as `type(uint32).max - (type(uint32).max % range) - 1` to ensure the accepted space contains exactly a multiple of `range`.

Impact

Any consumer treating `getRoundResultExtended` outputs as uniformly random can be economically biased against, especially in low-range selections (coin flips, small lotteries, tie-breakers). The attacker does not need privileged roles; simply calling the function produces outcomes with a slightly better-than-fair chance to land on the minimum option. While the bias is small (`1/2^32` per draw), it is:

- **Deterministic**: The bias is mathematically guaranteed, not probabilistic
- **Statistically verifiable**: Over large samples (e.g., 2000+ coin flips), the skew becomes measurable
- **Exploitable**: In scenarios where outputs drive allocations, selections, or payouts, this edge can be leveraged systematically

The vulnerability affects all range sizes, but the relative impact is most significant for small ranges where the bias represents a larger fraction of the outcome space.

Recommendation

It's recommended to change the comparison operator from strictly greater than (`>`) to greater than or equal to (`>=`).

```
uint32 range = max - min + 1;
uint32 maxValid = type(uint32).max - (type(uint32).max % range);

uint32 candidate = uint32(temp);
// Correct rejection sampling: reject candidate >= maxValid
if(candidate >= maxValid){
    continue;
}

newNumber = candidate % range + min;
```

Alleviation

[oncha, 12/07/2025]:
The issue has been fixed. Here is the new random number contract address:0xe0b676819a5C371670AAf2a231566c065e91beD1

ONC-01 | Signature Replay Attack

Category	Severity	Location	Status
Logical Issue	Informational	contracts/NFT.sol (NFT): 264~266	Acknowledged

Description

The signature for `_verifyAndProcess()` allows the same signature to be submitted multiple times before the `request.timestamp + VALIDITY_PERIOD`. As a result, the malicious `OPERATOR_ROLE` may transfer more `paymentToken` from user to current contract, which may not align with the user's expectation.

Recommendation

We would like to confirm if it is an intended design. If not, we recommend adding a nonce to the signature to avoid possible replay attacks. Additionally, it is recommended to include address(this) and the blockchain ID in the signed data, or utilize the EIP-712 domain separator, to prevent signature replay across different contracts or chains.

Alleviation

[oncha, 12/02/2025]:
We acknowledge the risk and no fix for this.

Explanations:

1. We are going to trust that the operator does not submit it multiple times.
2. In case of any duplicate requests due to bugs, we will notify users and compensate them.
3. We don't want to add a nonce because that adds more states on the contract, and we don't want to manage the nonce in the app.
4. We are aware of the fact that this signature can be replayed in other chains, but we will never deploy the contract in other chains.

ONC-02 | Missing `nftNumber` In Signed Message

Category	Severity	Location	Status
Design Issue	Informational	contracts/NFT.sol (NFT): 47~54, 260~262, 273	Acknowledged

Description

The `BuyRequest` struct includes an `nftNumber` field intended to identify the specific NFT being purchased:

```
struct BuyRequest {
    string nftType;
    address user;
    uint64 nftPage;
    uint64 nftNumber;
    uint64 nftAmount;
    uint64 timestamp;
}
```

However, in the `_verifyAndProcess` function, the `messageHash` used for signature verification does **not** include the `nftNumber`:

```
bytes32 messageHash = keccak256(
    abi.encode(request.nftType, request.nftPage, request.nftAmount,
request.timestamp)
);
```

This omission allows the account with `OPERATOR_ROLE` to call `_verifyAndProcess` using the **same valid user signature**, but with a **different** `nftNumber`, enabling the operator to arbitrarily alter which NFT number the user is purchasing. Since the signature does not bind `nftNumber`, the user has no cryptographic protection against such manipulation.

This could lead to:

- Users receiving NFTs different from what they intended to purchase.
- Operators being able to front-run or reassign valuable NFT numbers.
- Users signing a message that does not accurately reflect the full buy request being executed.

Recommendation

It is unclear whether this behavior is intentional; clarification is recommended.

Alleviation

[oncha, 12/02/2025]:
We acknowledge the risk and no fix for this.

Explanations:

Our user will not care about the nftNumber, and the nftNumber is assigned in the backend after collecting the user's signature.

ONC-03 | Unclear Integration Between `NFT` And `RandomNumber` Contracts

Category	Severity	Location	Status
Design Issue	Informational	contracts/NFT.sol (NFT): 13~14	Acknowledged

Description

The audit scope includes two main contracts within the current audit scope: `NFT` (a role-gated sales/payment contract) and `RandomNumber` (a VRF-based randomness provider). However, there is no observable on-chain integration between them:

- The `NFT` contract never calls `RandomNumber`, nor does it accept any randomness as an input parameter.
- The `RandomNumber` contract generates and exposes randomness, but no contract in scope appears to consume these values.

As a result, from the on-chain logic alone, it is not possible to determine:

- Whether randomness is actually used to drive NFT allocation, reward assignment, or purchase eligibility; or
- Whether the purchase and reward flows in `NFT` are intended to be randomness-dependent at all.

Recommendation

The audit team would like to clarify how the two in-scope contracts are intended to interact in the current system, and in particular whether the NFT reward distribution process is designed to be randomness-driven, either on-chain or off-chain.

Alleviation

[oncha, 12/02/2025]:
The RandomNumber contract and the NFT contract are two components. They are not directly connected on-chain.

The RandomNumber contract is to generate random numbers that users can trust that they will be truly random.

The NFT contract is for the NFT flow process. We just want to make sure there is no security hole in that contract.

ONC-11 | `fulfillRandomWords()` May Revert

Category	Severity	Location	Status
Volatile Code	Informational	contracts/RandomSeed.sol (RandomNumber): 128, 130	Acknowledged

Description

If the `fulfillRandomWords()` implementation reverts, the VRF service will not attempt to call it a second time, which may cause the corresponding request to remain unused. Make sure contract logic does not revert.

Recommendation

We would like to confirm if current implementation aligns with the intended design and recommend following the advice given by chainlink: https://docs.chain.link/vrf/v2/security.

Alleviation

[oncha, 12/02/2025]:
Yes, we are aware of the VRF service behavior, and the current behavior is fine for us.

ONC-12 | Potential Incorrect Function Visibility Or Naming Conventioin Violation

Category	Severity	Location	Status
Code Optimization	Informational	contracts/NFT.sol (NFT): 257	Resolved

Description

The current contract doesn't follow the naming convention specified by Solidity DOC, including the following:

- The function name starts with `_`. However, the function is publicly/externally visible. This violates the naming convention, or even exposes the internal function to the public, which is quite dangerous.

```
function _verifyAndProcess(BuyRequest calldata request, bytes calldata signature)
external onlyRole(OPERATOR_ROLE) {
  /// ...
}
```

Recommendation

To mitigate this issue, it is recommended to follow the naming conventions, including:

- Remove the underscore prefix for public functions or ensure internal functions are not exposed publicly.

Alleviation

[oncha, 12/02/2025]:

The team heeded the advice and resolved the issue in the updated version 0x5b2f1e8a6ba28b40acedc49af5ea66a91501325d.

ONC-13 | Debug And Test Logic In `RandomNumber` Contract Increases Complexity And Reduces Maintainability

Category	Severity	Location	Status
Coding Style	Informational	contracts/RandomNumber.sol (RandomNumber): 6	Acknowledged

Description

The `RandomNumber` contract includes several logic branches that are specifically intended for debugging or local testing. For example, when `useOracle` is disabled, the contract uses block data as a source of pseudo-randomness:

```
if (!useOracle) {
    // Use block data as pseudo-randomness (not secure, for testing only)
    uint256 l2BlockNumber = ArbSys(address(100)).arbBlockNumber();
    s_pendingRounds[round] = PendingRound({
        round: round,
        commitment: secretCommitment,
        vrfResult: 0,
        commitBlock: l2BlockNumber
    });
    s_roundFulfilledBlock[round] = l2BlockNumber;
    return 0;
}
```

This debug-oriented logic introduces additional branching and state handling that is not part of the production randomness workflow. The coexistence of both production and test-only logic increases the overall complexity of the contract, making it harder to audit, maintain, and reason about correctness. Moreover, the presence of testing logic in deployed code increases the risk of accidental misconfiguration (e.g., `useOracle` remaining disabled), which could degrade security by relying on insecure pseudo-randomness.

Recommendation

Separating production logic from debug/testing behavior will enhance readability, reduce accidental misuse, and improve long-term maintainability.

Alleviation

[oncha, 12/02/2025]:
We acknowledged this coding style issue and decided not to update the coding style for now.

We may actually use the pseudo-randomness configuration in certain low-risk cases.

ONC-14 | Unnecessary Fund VRF Consumer Contract

Category	Severity	Location	Status
Design Issue	Informational	contracts/RandomSeed.sol (RandomNumber): 235	Resolved

Description

The `RandomSeed` contract includes a `fundContract()` function that allows ETH to be deposited into the contract itself. However, when using Chainlink VRF v2+ subscriptions, request fees are deducted from the coordinator-owned subscription balance, not from the consumer contract's balance. The current implementation does not forward the received ETH to the subscription using `fundSubscriptionWithNative`, so depositing funds via `fundContract()` has no effect on the ability to pay for VRF requests. As a result, if the subscription's native balance is insufficient, VRF requests that require native payment will continue to revert with an “Insufficient balance” error, even after calling `fundContract()`. The presence of this function may confuse users regarding the proper way to fund VRF operations.

```
// Function to fund the contract with ETH for VRF requests
function fundContract() external payable onlyOwner {
    // Contract receives ETH to pay for VRF requests
}
```

Recommendation

It's recommended to remove `fundContract()` or replace it with a helper that actually tops up the subscription.

Alleviation

[oncha, 12/02/2025]:
The team heeded the advice and resolved the issue in the updated version 0x6643adcdb60ff832a7e0ec823572edffb6c08f5b.

ONC-17 | Potential Predictable Outputs In `getRoundResultExtended` Due To Seed Reuse Across Multiple Blocks/Transactions

Category	Severity	Location	Status
Design Issue	Discussion		Pending

Description

The function `getRoundResultExtended` derives multiple random numbers using a single round seed obtained from:

```
uint256 seed = getRoundResult(round);
```

This seed is constant for the entire round. The extended random number generation relies solely on deterministic hashing of predictable inputs:

```
uint256 temp = uint256(
    keccak256(abi.encode(name, seed, i + offset, attempts))
);
uint32 candidate = uint32(temp);
newNumber = candidate % range + min;
```

As a result:

1. **Once the seed becomes public (after round finalization), all future outputs are fully predictable.**
2. **If randomness-dependent actions are allowed after the seed is revealed**, an attacker can:
    - simulate all outputs offline,
    - compute all candidate values,
    - front-run transactions,
    - or exploit predictable ordering or selection logic.
3. **If the system allows multiple blocks or transactions to use the same round seed**, the window in which attackers can predict and exploit outcomes grows significantly.

Because the only entropy source is the round seed — and all other inputs (`name` , `i` , `offset` , `attempts`) are predictable or attacker-controlled — the randomness derived in `getRoundResultExtended` becomes entirely deterministic and easily precomputable.

This issue is especially impactful if `getRoundResultExtended` is used for:

- NFT trait assignment,
- lottery or winner selection,

- shuffling or ordering,
- game logic,
- or any mechanism where randomness determines valuable outcomes.

Recommendation

We would like to confirm if the above case has been considered.

APPENDIX | ONCHA

Finding Categories

Categories	Description
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.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
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.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

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.

Elevating Your Web3 Journey

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is the largest 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.