SECURITY AUDIT OF

CRABADA SALE CONTRACTS

Public Report

Nov 03, 2021

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report prepared by Verichains Lab on Nov 03, 2021. We would like to thank the CRABADA for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Crabada sale contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the application.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Crabada sale contracts

A Fully DecentralizedPlay-and-Earn Idle Game

Rediscover the prosperous ancient Crabada Kingdom once ruled by Crustaco, King of the Crabada.

Mine. Loot. Breed. Expand your forces.

Earn CRA tokens by playing and use them to determine the future of the Kingdom!

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of Crabada sale contracts. It was conducted on the source code provided by the CRABADA team.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The initial review was conducted on Oct 25, 2021 and a total effort of 4 working days was dedicated to identifying and documenting security issues in the code base of the Crabada sale contracts.

The following files were made available in the course of the review:

SHA256 SUM	FILE
300eb199a63c4d4b20db97d000c8ab66c9d4847d1a8341891be1e5cb0e46d45c	CrabadaSale.sol
4fd6092bdfa8b42f19d535c5ac69c4323b0b894717c699e58d5552eeabd04cd4	Migrations.sol
1660d29e5f5c3bcf0e3b18b297f370ca05dbc513a9eb05a09226ceb4eb8b26bb	BreedingCoin.sol
d80f91e3fcb234d27c6f93837cf9058661fed3f3683cfd8a9dd506579b5b5bff	Chamber.sol
b421287ac571312e7deb337f31842c5ba03ec1b6d35c2f343ada8c8798bb8ec5	Distributor.sol
d97f17fcf78173ebdf34205f4ecf20659b746043a0c8816b9431ec0f289003e2	FundKeeper.sol
330b94d641c9605f81822ee52f70c4ea87c92a084cc1d4c83e7a7a7c505b59a1	Timelock.sol

2.2. Findings

The Crabada sale contracts was written in Solidity language, with the required version to be ^0.8.0. The source code was written based on OpenZeppelin's library.

CRABADA fixed the code according to Verichain's draft report

The contracts use third party contract (randomGenerator) to generate random number, and we don't have access to that service, so we only assume that randomGenerator.getRandomNumber() will return a random uint256. We can't check any vulnerabilities relate to RNG in this audit process.

2.2.1. Chamber.sol - Missing non-contract call checking CRITICAL

The buy and buyPure functions are missing non-contract call checking. If functions using RNG does not block contract call, attackers can use contract to call-revert to control the result of random and get the best value.

```
function buy() nonReentrant() whenNotPaused() checkMaxCrabPerAddress(msg.…
  sender) external {
    ...
    uint256 randomness = randomGenerator.getRandomNumber();
    ...
}

function buyPure() nonReentrant() whenNotPaused() external {
    ...
    uint256 randomness = randomGenerator.getRandomNumber();
    ...
}
```

RECOMMENDATION

Adding onlyNonContract modifier to all functions which use the randomGenerator.getRandomNumber() function.

```
modifier onlyNonContract {
    require(tx.origin == msg.sender, "Only non-contract call");
    _;
}
```

UPDATES

- *2021-11-01*: This issue has been acknowledged and fixed by the CRABADA team.

2.2.2. Timelock.sol - Wrong time checking for onlyStarted modifier LOW

Modifier onlyStarted will be wrong if startDate set to future in startSchedule function.

```
modifier onlyStarted() {
    require(startDate > 0, "NOT START");
    _;
}
```

```
function startSchedule(uint256 _startDate) onlyOwner() external {
    require(startDate == 0, "STARTED");

    if (_startDate == 0) {
        startDate = block.timestamp;
    } else {
        startDate = _startDate;
    }

    emit Start(startDate);
}
```

RECOMMENDATION

Add startDate < block.timestamp to require.

```
modifier onlyStarted() {
    require(startDate > 0 && startDate < block.timestamp`, "NOT START");
    _;
}
```

UPDATES

- *2021-11-01*: This issue has been acknowledged and fixed by the CRABADA team.

2.2.3. CrabadaSale.sol - Wrong remainTokenAmount if change totalSaleAmount while sale is running INFORMATIVE

Change totalSaleAmount by calling changeTotalSaleAmount while sale is running (when remainTokenAmount > 0) will make remainTokenAmount wrong.

```
function changeTotalSaleAmount(uint256 newAmount) onlyOwner() external {
    totalSaleAmount = newAmount;
    remainTokenAmount = newAmount;
}
```

RECOMMENDATION

Avoid set totalSaleAmount while sale is running.

```
function changeTotalSaleAmount(uint256 newAmount) onlyOwner() external {
    require(remainTokenAmount == 0, 'Sale is running'); // Add this
    require(newAmount != totalSaleAmount, 'Amount not change'); // Add th…
  is
    totalSaleAmount = newAmount;
    remainTokenAmount = newAmount;
}
```

UPDATES

- *2021-11-01*: This issue has been acknowledged by the CRABADA team.

2.2.4. Gas optimize INFORMATIVE

Using memory argument will force Solidity to copy it to memory which will cost more gas than using from calldata especially when passing large readonly array.

RECOMMENDATION

Consider change memory to calldata in all contracts for gas saving.

Example:

```
function addWhitelist(address[] calldata addrs, uint256[] calldata allocs…
  ) onlyOwner() external {
    uint256 totalAlloc;
    for (uint256 i = 0; i < addrs.length; i++) {
        whitelist[addrs[i]] = true;
        allocations[addrs[i]] = allocs[i];
        totalAlloc += allocs[i];
    }
    whitelistAllocationTotal += totalAlloc;
    whitelistCount += addrs.length;
}
```

UPDATES

- *2021-11-01*: This issue has been acknowledged and fixed by the CRABADA team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*2021-10-29*	Private Report	Verichains Lab
1.1	*2021-11-01*	Private Report	Verichains Lab
1.2	*2021-11-03*	Public Report	Verichains Lab

Table 2. Report versions history