QuillAudits

Audit Report
November, 2022

For

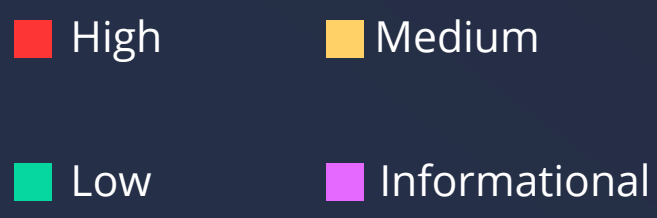
CR SQUARE

Table of Content

Table of Content

Executive Summary

Project Name	CR2
Overview	A next generation funding platform which provides a safe route of investment for Investors to invest in next generation web3 startups. Founders can make connections with investors and raise funds based on completion of pre-determined milestones. New projects and ideas get full funding, while at the same time protecting the investors from misuse of the investment.
Timeline	20th October, 2022 to 24 November, 2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyze CR2's codebase for quality, security, and correctness. *https://github.com/CR-Square/crsquareq/tree/main/contracts* Commit hash: 9b57d74e439f760c89d6ca695e99191d0da80888
Fixed In	2c1339fba6ba598f79be47c858aa10b2f8b29526



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	1
Partially Resolved Issues	0	0	0	0
Resolved Issues	3	0	8	8

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

High Severity Issues

A1. Hypercritical functions design flaw

Path	contracts/Vesting.sol
Function	depositFounderLinearTokens/ depositFounderLinearTokensToInvestors/ depositFounderNonLinearTokens/ withdrawTGEFund/ withdrawInstallmentAmount/ withdrawBatch/ setNonLinearInstallments

Background

All users are allowed to become investors and founders at the same time. This is a straightforward, trustable, and seemingly harmless entity model for a custom vesting protocol. However, the implementation in the Vesting contract does not place appropriate data validations, while executing deposits and withdrawals.
User controlled inputs functions always allow users for a rational execution, a large number of inputs means a lot of attack surface for an adversary. This practice could still be directed by placing apt constraints against the wide array of inputs which in other cases, leads to exploits in smart contracts like DoS, Storage manipulation, overwriting vital lookup mappings etc.
The current codebase opens up such a window of potential adversarial scenarios, where an attacker can fracture the protocol completely; in terms of integrity, authentication, and nonrepudiation.

Description

An attacker can become a legitimate user after registering into the contracts namely Founder and InvestorLogin then make arbitrary deposits as early as possible to bypass the timelock conditions for withdrawable positions via two possible scenarios:

1) Calling addFounder and addInvestor on legit contracts of the protocol respectively or,
2) Creating fake instances for Founder and InvestorLogin contracts with functions verifyFounder and verifyInvestor that always return true.

Following this setup the attacker can maliciously utilize the protocol such as;
1) Being an investor:

a) Frontrun legit withdrawals called by investors of worthwhile tokens, though with the same amounts but different _symbol for the above valuable token.
b)Call withdraw functions on behalf of other investors by providing _symbol(s) of legit tokens along with the initially set up bloated amounts of the fake coin, soon after the _tgeFund is unlocked.

PoC of a critical severity (an example situation)
An attacker being an investor can withdraw against worthy tokens' investors by following steps below:
(i) Register into InvestorLogin by calling addInvestor
(ii) Make deposit of an arbitrarily large amount
(iii) Call setNonLinearInstallments with correct inputs
(iv) Track other legit investors with big amounts of worthwhile tokens
(v) Frontrun investor's withdrawals who have high token value after the vesting period is over with the same amount by passing their _symbol token\

Being a founder:
(a) Manipulate vestingDues details of other founders by overwriting amounts of token with founder's fake token
(b) Re Enter into different functions using ERC20 transfer and transferFrom to drain tokens, due to reasons mentioned in issue no 8 below;
(c) Frontrun setting up installment plans of other founders having legitimate tokens with arbitrary due _dates, _status and _funds
(d) Can overwrite vestID against worthy tokens' investors by following steps below;
Here the attack opens up when this attacker breaks the integrity of the deposit by manipulating the _vestID of investors to lead to DOS and storage manipulation attacks.
(i) Consider the scenario when a malicious founder calls a deposit for a fake token with arbitrary _amount and _tgeFund to maliciously update the vestingDues mapping with the correct _vestID and _investor.
(e) Call setNonLinearInsallments against any current vesting program with arbitrary _vestID
(i)Unconstrained setting of installments plan

Code

```
// Below function signatures contain the insufficient input validations leading to a critically severe attack surface.

 function depositFounderLinearTokens(uint _tgeFund, founderSetup memory _f, bytes32 _symbol, uint _vestId, uint _amount, address _investor, uint _tgeDate, uint _vestingStartDate, uint _vestingMonths, uint _vestingMode) external {

  function depositFounderNonLinearTokens(address _founder, address _founderCoinAddress, address _founderSmartContractAd, bytes32 _symbol, uint _vestId, uint _amount, address _investor, uint _tgeDate, uint _tgeFund) external{
```

```
  function withdrawTGEFund(address _investor,address _founder, uint _vestId,
bytes32 _symbol) external {

  function setNonLinearInstallments(address _founder, address
_founderSmartContractAd, uint _vestId, address _investor,due[] memory _dues)
external {
```

Remediation

For Vesting contract:
Restructure the following external functions focusing on function arguments and input validations to reduce overall attack surface:
Method ID => Function Signature
49839260 => depositFounderNonLinearTokens(address,address,address,bytes32,uint256,uint256,address,uint256,uint256)

974f4213 => depositFounderLinearTokens(uint256,founderSetup,bytes32,uint256,uint256,address,uint256,uint256,uint256,uint256)

be8add79 => depositFounderLinearTokensToInvestors(forFounder,bytes32,uint256,uint256,uint256,uint256,investors[],uint256)

0793a56c => withdrawTGEFund(address,address,uint256,bytes32)

cc6c0a22 => withdrawInstallmentAmount(address,address,uint256,uint256,bytes32)

72b43e03 => withdrawBatch(address,address,uint256,bytes32)

e260c561 => setNonLinearInstallments(address,address,uint256,address,due[])

Status

Resolved

A2. Inexistent logic to verify investors and RoundId lead to malicious withdrawl of funds

Path	contracts/PrivateRound.sol
Function	validateMilestone

Description

Founders can withdraw funds after investors have validated the milestone. Current implementation allows anyone to validate milestones.

Attack Scenario
Founders will deploy a fake InvestorLogin contract to validate the milestones

Code

```
function validateMilestone(address _investorSM, uint _milestoneId, uint _roundId,
bool _status) external {
      require(msg.sender != address(0), "The address is not valid or the address is 0");
      require(_investorSM != address(0), "The smart contract address is invalid");
      InvestorLogin investor = InvestorLogin(_investorSM); //@audit user controlled
address
      require(investor.verifyInvestor(msg.sender), "The address is not registered in the
'InvestorLogin' contract");
      if(milestoneApprovalStatus[_roundId][_milestoneId] == 1){
         revert("The milestone is already approved");
      }
      if(_status){
         milestoneApprovalStatus[_roundId][_milestoneId] = 1;
      }else{
         rejectedByInvestor[_roundId][_milestoneId] += 1;
         milestoneApprovalStatus[_roundId][_milestoneId] = -1;
      }
      if(rejectedByInvestor[_roundId][_milestoneId] >= 3){
         projectCancel[_roundId]

= true;
      }
   }
```

Remediation

InvestorLogin smart contract address should be stored in contract state variable by admins. Change the InvestorLogin contract to hold/whitelist all investors' addresses. Also properly verify the roundId is for a specific Investor.

Status

Resolved

A3. Potential reentrancy in withdrawTaxTokens

Path	contracts/PrivateRound.sol
Function	withdrawTaxTokens

Description

Admins can withdraw taxed tokens. The amount they can withdraw is stored in taxedTokens[_tokenContract]. ERC777 tokens allow reentrancy, admins can call withdrawTaxTokens multiple times leading to potential rug pull.

Code

```
function withdrawTaxTokens(address _tokenContract, uint _roundId, address
_founder) external onlyAdmin {
      require(msg.sender != address(0), "Invalid address"); //@audit invalid check
      FundLock fl = FundLock(seperateContractLink[_roundId][_founder]);
      require(ERC20(_tokenContract).transferFrom(address(fl), msg.sender,
taxedTokens[_tokenContract]), "execution failed or reverted");
      taxedTokens[_tokenContract] = 0; //here
   }
```

Remediation

We recommend to update taxedTokens[_tokenContract] = 0 before transferring tokens.

Status

Resolved

Low Severity Issues

A4. Potential front run in createPrivateRoun

Path	contracts/PrivateRound.sol
Function	createPrivateRound

Description

While calling createPrivateRound function, the user provides desired _roundID as input which create an opportunity for an adversary (e.g. a malicious investor) to always front-run other _investor to create a false _roundID resulting in transaction failure each time ultimately lead to Denial-of-Service (DoS) attack.

Code

```
function createPrivateRound(uint _roundId, address _investorSM, uint
_initialPercentage, MilestoneSetup[] memory _mile) external {
      require(msg.sender != address(0), "The address is not valid or the address is 0");
      require(_investorSM != address(0), "The contract address is invalid or address
0");
      InvestorLogin investor = InvestorLogin(_investorSM);
      require(investor.verifyInvestor(msg.sender), "The address is not registered in the
'InvestorLogin' contract");
      if(roundIdControll[_roundId]){
        revert("round Id is already taken");
      }
      for(uint i = 0; i < _mile.length; ++i){
        _milestone[msg.sender][_roundId].push(_mile[i]);
        milestoneApprovalStatus[_roundId][_mile[i]._num] = 0;
```

Remediation

Implement a mechanism to increment _roundID controlled by the smart contract instead of asking the user as a function argument.

Status

Resolved

A5. WhiteList tokens to prevent unwanted behavior of contract

Path	contracts/PrivateRound.sol
Function	depositTokens

Description

As there are negligible constraints in the function above of the PrivateRound smart contract, users can provide any type of _tokenContract to create a private round, which opens up a bunch of opportunities for an attacker and at the very least, a window of execution for a corrupt actor using malicious ERC20 tokens. Since, allowing interactions with an arbitrary token can lead to unwanted behaviors within the smart contract.

Code

```
function depositTokens(address _tokenContract, address _investorSM, address
_founder, uint _tokens, uint _roundId) external { }
```

Remediation

Implement a mechanism to whitelist tokens and restrict users to only use whitelisted tokens to create rounds and deposit tokens.

Status

Resolved

A6. Gas optimized checks to revert

Path	contracts/PrivateRound.sol
Function	depositFounderLinearTokens/ depositFounderLinearTokensToInvestors/ depositFounderNonLinearTokens/ withdrawTGEFund/ withdrawInstallmentAmount/ withdrawBatch/ setNonLinearInstallments

Description

Custom errors are available from solidity version 0.8.4. Multiple instances throughout the codebase utilize the require statements to place constraints and revert otherwise. This is a costly operation proven over time and can be optimized as follows.

Remediation

This operation can be optimized by using custom error messages in combination with if blocks, simply mirroring the conditions in require statement to place in if block and reverting when the condition is true with a message that could be self explanatory or might also contain a string message explaining the error.

Status

Resolved

A7.Fix investerLogin and Founder contract address

Path	contracts/PrivateRound.sol
Function	createPrivateRound/ depositTokens/ withdrawInitialPercentage/ milestoneValidationRequest/ validateMilestone/ withdrawIndividualMilestoneByFounder/ withdrawIndividualMilestoneByInvestor/ batchWithdrawByInvestors

Description

As there are negligible constraints in the function above of the PrivateRound smart contract, users can provide any type of _tokenContract to create a private round, which opens up a bunch of opportunities for an attacker and at the very least, a window of execution for a corrupt actor using malicious ERC20 tokens. Since, allowing interactions with an arbitrary token can lead to unwanted behaviors within the smart contract.

Code

```
function depositTokens(address _tokenContract, address _investorSM, address
_founder, uint _tokens, uint _roundId) external {
    require(msg.sender != address(0), "The address is not valid or the address is 0");
    require(_investorSM != address(0), "The smart contract address is invalid");
    require(_tokenContract != address(0), "The smart contract address is invalid");
```

Status

Resolved

A8. Missing isInitialized modifier

Path	contracts/PrivateRound.sol

Description

Since contract implements the upgradable UUPS standard initialized function, it is very important to be called right after the deployment, however it’s necessary to check before calling another function if the contract is initialized or not.

Remediation

Create a modifier of isInitialized and place it to every critical function which will affect the smart contract complex.

Status

Resolved

A9. Inexistent transfer of ownership

Path	contracts/PrivateRound.sol
Function	changeAdminAddress

Description

It is recommended to use two step transfer of ownership
• SECURITY: It allows the transfer of material function controls to a provably functional account.
LEGAL: It represents an on-chain "offer" and "acceptance" and formation of a legal contract that defensibly transfers smart contract ctrl.

Old Code:

```
function changeAdminAddress(address _newAdmin) external onlyAdmin{
    require(msg.sender != address(0), "Invalid address");
    require(_newAdmin != address(0), "The wallet address is invalid or address 0");
    contractOwner = _newAdmin;
}
```

New Code:

```
function grantOwnership(address newOwner) public virtual onlyOwner {
    emit OwnershipGranted(newOwner);
    _grantedOwner = newOwner;
}

function claimOwnership() public virtual {
    require(_grantedOwner == _msgSender(), "Ownable: caller is not the granted
owner");
    emit OwnershipTransferred(_owner, _grantedOwner);
    _owner = _grantedOwner;
    _grantedOwner = address(0);
}
```

Status

Resolved

A10. Improper implementation of state variables

Path	contracts/PrivateRound.sol

Description

• PrivateRound contract has 2 state variables 1) tokenContract 2) defaultedByFounder tokenContract variable changes its state whenever depositTokens function is called but that variable is never used throughout the function, so it does not affect any other functionality.

• Similarly, defaultedByFounder is also a state variable but is never used anywhere else, so there is no point to create a state variable and update it every time

Status

Resolved

A11. Data validation missing on addFounder/addInvestor functions

Path	contracts/PrivateRound.sol
Function	addInvestor/ addFounder

Description

In contract Founder and InvestorLogin there is a function addFounder and addInvestor which doesn't check if the founder or the investor already exist in the array.

Code

```
function addInvestor(address _ad) external{
    require(msg.sender == _ad,"Connect same wallet to add 'Investor address' ");
    isInvestor[_ad] = true;
    pushInvestors.push(_ad);
  }
```

Remediation

Create a modifier of isInitialized and place it to every critical function which will affect the smart contract complex.

Status

Resolved

Informational Issues

A12. Ensure secure deployment

Path	contracts/PrivateRound.sol

Description

The protocol being upgradeable uses openzeppelin's UUPS upgradeability patterns which require steady deployment and timely initialization to avoid frontrun or exploits made through transaction order dependence.

Remediation

It is highly recommended that the protocol deployment must be handled by a Javascript based scripting file which securely contains the keys for mainnet deployment and the correct order of deployment and initialization of appropriate code segments.

Status

Acknowledged

A13. Increase code readability

Path	contracts/PrivateRound.sol

Description

In the Vesting contract, there are multiple instances where readability can be improved by implying to the following recommendations in general

Remediation

- Redundant checks increase code size
  - Deposit functions place different checks that achieve the same idea and can be categorized into a modifier that can be reused for a cleaner look.
- Unoptimized structs usage
- Multiple structs are scattered throughout the file Vesting.sol which should be categorized following the solidity docs guide as mentioned in one of the issues above.

Absent cosmetic linting

- Static tools like Prettier and Solidity extension for VSCode can be used so that whitespaces and carriages are completed.

Order of layout; functions and event

- Following the recommendations in issue above for the Order of Layout as provided in Solidity Docs guide.
- Inconsistent naming (variables, structs, mappings)

Throughout codebase naming of state and local variables, structs and mappings is highly inadequate which leads to confusion in misunderstandings in code readability and ultimately execution.

Status

Resolved

A14. Use structs Function Parameters

Path	contracts/Vesting.sol
Function	depositFounderLinearTokens/ depositFounderLinearTokensToInvestors/ depositFounderNonLinearTokens/ withdrawTGEFund/ withdrawInstallmentAmount/ withdrawBatch/ setNonLinearInstallments

Description

In the Vesting contract the above mentioned functions accept inputs in an ad hoc manner which can be confusing for users or DApps composing utilizing this smart contracts. In which it’s checking whether msg.sender != address(0), however the check is irrelevant in any case the msg.sender can not be 0 address.

Recommendation

s(0), It is highly recommended that functions be restructured to accept parameters as structs so that variables during the execution of function keep organized. For instance, DepositLinearFTKToInvestorsStruct can be used for the function depositFounderLinearTokensToInvestors

Status

Resolved

A15. Illogical use of require check

Path	contracts/PrivateRound.sol
Function	createPrivateRound/ depositTokens/ withdrawInitialPercentage/ milestoneValidationRequest/ validateMilestone/ withdrawIndividualMilestoneByFounder/ withdrawIndividualMilestoneByInvestor/ batchWithdrawByInvestors/ changeAdminAddress/ withdrawTaxTokens

Description

In the contract PrivateRound many functions have a require statement in which it's checking whether msg.sender != address(0), however the check is irrelevant as the msg.sender can never be the 0 address.

Code

```
function createPrivateRound(uint _roundId, address _investorSM, uint
_initialPercentage, MilestoneSetup[] memory _mile) external {
    require(msg.sender != address(0), "The address is not valid or the address is 0");
//here
    require(_investorSM != address(0), "The contract address is invalid or address
0");
```

Remediation

Remove the require statement from the function.

Status

Resolved

A16. Improper ordering of round ID

Path	contracts/PrivateRound.sol
Function	createPrivateRound

Description

The createPrivateRound function asks the user for the roundId in which the user can input any id which can collide sometimes if the roundId is already taken. It is an anti-pattern for smart contracts and an inefficient way for functions to ask the user the roundID everytime.

Code

```
function createPrivateRound(uint _roundId, address _investorSM, uint
_initialPercentage, MilestoneSetup[] memory _mile) { }
```

Remediation

Create a variable of roundId and increment it every time the user calls the createRound function.

Status

Resolved

A17. Inconsistent order of layout

Path	contracts/PrivateRound.sol; contracts/Vesting.sol

Description

As stated in the Solidity style guide, the contract components should be grouped according to their order and visibility:

Outside Contract

- License statement
- Pragma statement
- Import statemen
- Interfaces
- Libraries
- Contract statement

Inside Contract

- State Variables
- Events
- Function Modifiers
- Arrays
- Structs
- Enums
- Mappings
- constructor
- receive function (if exists)
- fallback function (if exists)
- External variable functions
- Public variable functions
- Internal variable functions
- Private variable functions

Status

Resolved

A18. Missing Natspec

Path	PrivateRound.sol

Description

There is no natural specification to elaborate the functionality of functions. NatSpec is an essential component of the overall codebase, being necessary to understand the core logic written by developers for users. It is recommended to place relevant NatSpec documentation explaining the assumptions of developers for every function for a clearer understanding of functionality.

Status

Resolved

A19. Inadequate Function calling

Path	contracts/PrivateRound.sol
Function	depositTokens/ withdrawInitialPercentage/ milestoneValidationRequest/ validateMilestone

Description

The functions name mentioned in the above table are callable by anyone even if the investor has not created a private round which sets mapping of _milestones, further connected or sequential functions should check if the private round is created otherwise no function should be callable.

Status

Resolved

A20. Inconsistent order of layout

Path	contracts/PrivateRound.sol; contracts/Vesting.sol

Description

No function in PrivateRound contract emits a relevant event, Events and logs are important in contracts because they facilitate communication between smart contracts and their user interfaces, Events notify the applications about the change made to the contracts and applications which can be used to execute the dependent logic.

Remediation

It is highly recommended to implement events in every critical function to read changes outside of blockchain.

Status

Resolved

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the CR2 codebase. We performed our audit according to the procedure described above.

Some issues of High, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the CR2 Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the CR2 Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

700+
Audits Completed

$15B
Secured

700K
Lines of Code Audited

Follow Our Journey

Audit Report
November, 2022

For

CR SQUARE

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com