BEOSIN

Blockchain Security

Smart contract security audit report

Audit Number：202101191847

Report Query Name：RAMP_PRIVATESALE_VESTING

Audit Project Name：RAMP_PRIVATESALE_VESTING

Audit Project Contract Info：

Project URL	https://github.com/RAMP_PRIVATESALE_VESTING/RAMP_PRIVATESALE_VESTING/blob/main/flattened/PrivateSaleVesting.sol
Origin audit commit id	d16685cc985e2c7b78a34983437f7b16af39da97
Final audit commit id	872b465945445a7bb177cd581a70704a0c1c0beb

Start Date：2021.01.14

Completion Date：2021.01.19

Overall Result：Pass

Audit Team: Beosin (Chengdu LianAn) Technology Co. Ltd.

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	Compiler Version Security	Pass
		Deprecated Items	Pass
		Redundant Code	Pass
		SafeMath Features	Pass
		require/assert Usage	Pass
		Gas Consumption	Pass
		Visibility Specifiers	Pass
		Fallback Usage	Pass
2	General Vulnerability	Integer Overflow/Underflow	Pass
		Reentrancy	Pass
		Pseudo-random Number Generator (PRNG)	Pass
		Transaction-Ordering Dependence	Pass
		DoS (Denial of Service)	Pass

		Access Control of Owner	Pass
		Low-level Function (call/delegatecall) Security	Pass
		Returned Value Security	Pass
		tx.origin Usage	Pass
		Replay Attack	Pass
		Overriding Variables	Pass
3	Business Security	Business Logics	Pass
		Business Implementations	Pass

Note: Audit results and suggestions in code comments

Disclaimer: This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin (Chengdu LianAn) Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin (Chengdu LianAn) Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin (Chengdu LianAn) Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin (Chengdu LianAn) Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin (Chengdu LianAn) Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin (Chengdu LianAn). Due to the technical limitations of any organization, this report conducted by Beosin (Chengdu LianAn) still has the possibility that the entire risk cannot be completely detected. Beosin (Chengdu LianAn) disclaims any liability for the resulting losses.

The final interpretation of this declaration belongs to Beosin (Chengdu LianAn).

Audit Results Explained:

Beosin (Chengdu LianAn) Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of smart contracts project RAMP_PRIVATESALE_VESTING, including Coding Standards, Security, and Business Logic. **The RAMP_PRIVATESALE_VESTING project passed all audit items. The overall result is Pass.** The smart contract is able to function properly.

Audit Contents:

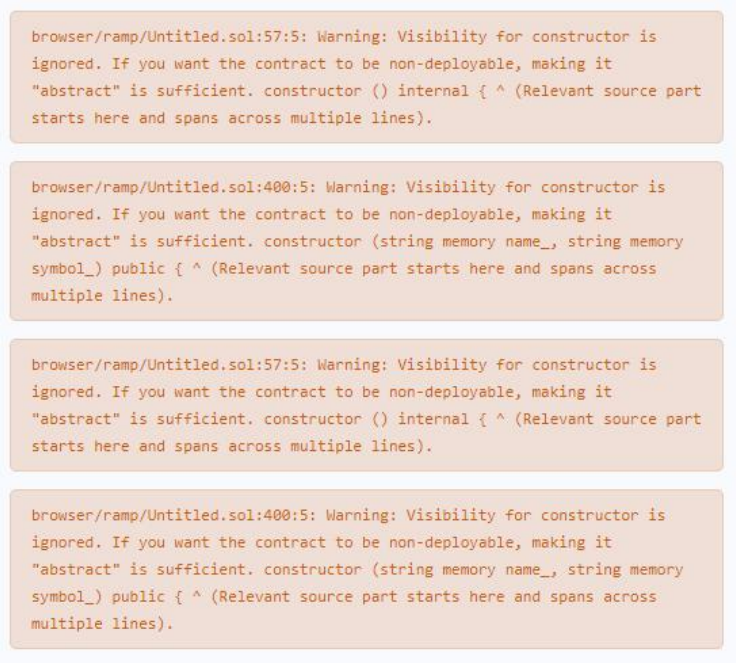
1. Coding Conventions

Check the code style that does not conform to Solidity code style.

1.1 Compiler Version Security

- Description: Check whether the code implementation of current contract contains the exposed solidity compiler bug.

  The contract of this project specifies that the minimum compiler version of the contract is 0.7.0. When the contract is compiled with this version of the compiler, there is are some compiler warning as shown in the figure below:



```
browser/ramp/Untitled.sol:57:5: Warning: Visibility for constructor is
ignored. If you want the contract to be non-deployable, making it
"abstract" is sufficient. constructor () internal { ^ (Relevant source part
starts here and spans across multiple lines).

browser/ramp/Untitled.sol:400:5: Warning: Visibility for constructor is
ignored. If you want the contract to be non-deployable, making it
"abstract" is sufficient. constructor (string memory name_, string memory
symbol_) public { ^ (Relevant source part starts here and spans across
multiple lines).

browser/ramp/Untitled.sol:57:5: Warning: Visibility for constructor is
ignored. If you want the contract to be non-deployable, making it
"abstract" is sufficient. constructor () internal { ^ (Relevant source part
starts here and spans across multiple lines).

browser/ramp/Untitled.sol:400:5: Warning: Visibility for constructor is
ignored. If you want the contract to be non-deployable, making it
"abstract" is sufficient. constructor (string memory name_, string memory
symbol_) public { ^ (Relevant source part starts here and spans across
multiple lines).
```

Figure 1 Compiler warning for this contract

- Safety Recommendation: Delete the visibility declaration of each contract constructor.
- Fix Result: Ignored. The sub-contracts of this contract are directly referenced in the openzeppelin library; and have no impact on the contract business logic.
- Result: Pass

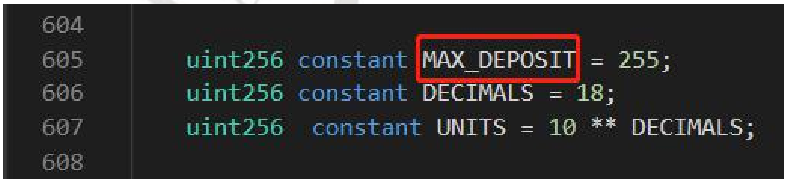
1.2 Deprecated Items

- Description: Check whether the current contract has the deprecated items.
- Safety Recommendation: None
- Result: Pass

1.3 Redundant Code

- Description: Check whether the contract code has redundant codes.

As shown in the figure below, the MAX_DEPOSIT constant is declared in the contract, but it is not used in the contract.



```
    uint256 constant MAX_DEPOSIT = 255;
    uint256 constant DECIMALS = 18;
    uint256  constant UNITS = 10 ** DECIMALS;
```

Figure 2 MAX_DEPOSIT variable declaration

- Safety Recommendation: Delete the MAX_DEPOSIT constant.
- Fix Result: Fixed. Has been modified to MAX_TRANCHES, and used in the *addTranche* function.
- Result: Pass

1.4 SafeMath Features

- Description: Check whether the SafeMath has been used. Or prevents the integer overflow/underflow in mathematical operation.
- Safety Recommendation: None
- Result: Pass

1.5 require/assert Usage

- Description: Check the use reasonability of 'require' and 'assert' in the contract.
- Safety Recommendation: None
- Result: Pass

1.6 Gas Consumption

- Description: Check whether the gas consumption exceeds the block gas limitation.

According to the contract logic, the *_withdrawable* function will traverse each tranche in the system when user calling the *withdraw* function to withdraw token. Therefore, if there are a large number of tranches in the system, the call of the *withdraw* function will fail because the gas exceeds the gas limit.

- Safety Recommendation: Limit the number of tranche to avoid too many loops in *_withdrawable* function.
- Fix Result: Fixed.
- Result: Pass

1.7 Visibility Specifiers

- Description: Check whether the visibility conforms to design requirement.
- Safety Recommendation: None
- Result: Pass

1.8 Fallback Usage

- Description: Check whether the Fallback function has been used correctly in the current contract.
- Safety Recommendation: None
- Result: Pass

2. General Vulnerability

Check whether the general vulnerabilities exist in the contract.

2.1 Integer Overflow/Underflow

- Description: Check whether there is an integer overflow/underflow in the contract and the calculation result is abnormal.
- Safety Recommendation: None
- Result: Pass

2.2 Reentrancy

- Description: An issue when code can call back into your contract and change state, such as withdrawing ETH.
- Safety Recommendation: None
- Result: Pass

2.3 Pseudo-random Number Generator (PRNG)

- Description: Whether the results of random numbers can be predicted.
- Safety Recommendation: None
- Result: Pass

2.4 Transaction-Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Safety Recommendation: None
- Result: Pass

2.5 DoS (Denial of Service)

- Description: Whether exist DoS attack in the contract which is vulnerable because of unexpected reason.
- Safety Recommendation: None
- Result: Pass

2.6 Access Control of Owner

- Description: Whether the owner has excessive permissions, such as malicious issue, modifying the balance of others.
- Safety Recommendation: None
- Result: Pass

2.7 Low-level Function (call/delegatecall) Security

- Description: Check whether the usage of low-level functions like call/delegatecall have vulnerabilities.
- Safety Recommendation: None
- Result: Pass

2.8 Returned Value Security

- Description: Check whether the function checks the return value and responds to it accordingly.
- Safety Recommendation: None
- Result: Pass

2.9 tx.origin Usage

- Description: Check the use secure risk of 'tx.origin' in the contract.
- Safety Recommendation: None
- Result: Pass

2.10 Replay Attack

- Description: Check whether the implement possibility of replay Attack exists in the contract.
- Safety Recommendation: None
- Result: Pass

2.11 Overriding Variables

- Description: Check whether the variables have been overridden and lead to wrong code execution.
- Safety Recommendation: None
- Result: Pass

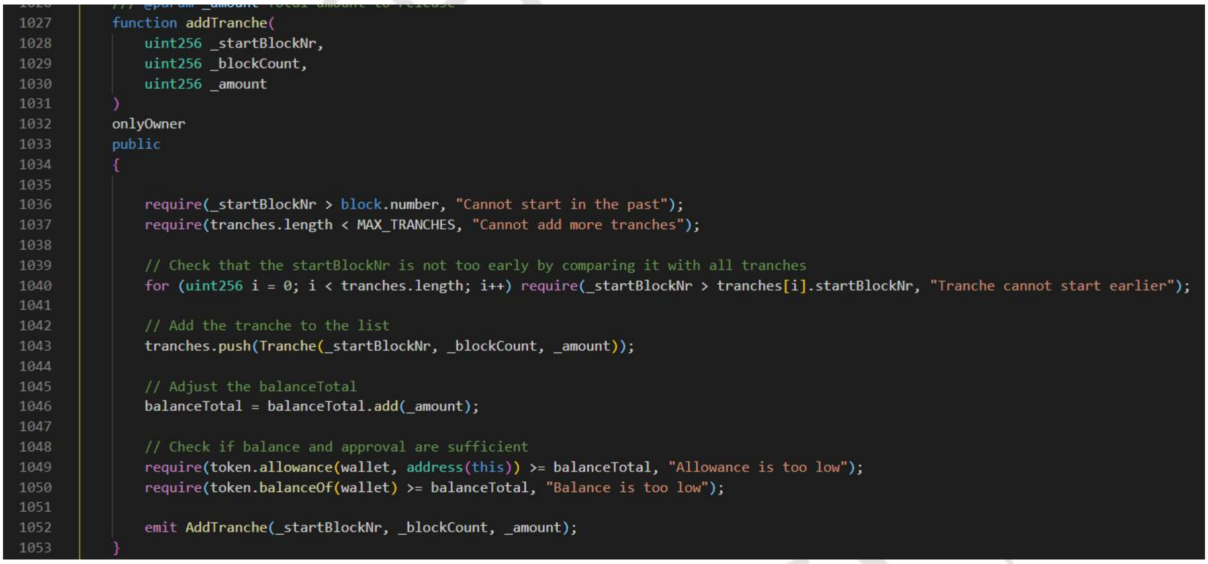
3. Business Audit

3.1 Contract management

- Description: The main contract PrivateSaleVesting inherits the Ownable contract. The contract manager owner(default is the contract deployer) can transfer the owner's permission to other non-zero addresses by calling the *transferOwnership* function. The *renounceOwnership* function can also be called to renounce the owner permission.
- Related functions: *transferOwnership, renounceOwnership*
- Safety Suggestion: None
- Result: Pass

3.2 Add tranche

- Description: The contract owner can call the *addTranche* function to add a tranche. As shown in the figure below, the function checks the validity of the input parameters, and requires that the number of tranche cannot be greater than 255; then adds tranche information; finally checks whether the wallet address's allowance to this contract and the current balance meet the user's allocation.



```
    /// @param _amount Total amount to release
    function addTranche(
        uint256 _startBlockNr,
        uint256 _blockCount,
        uint256 _amount
    )
    onlyOwner
    public
    {

        require(_startBlockNr > block.number, "Cannot start in the past");
        require(tranches.length < MAX_TRANCHES, "Cannot add more tranches");

        // Check that the startBlockNr is not too early by comparing it with all tranches
        for (uint256 i = 0; i < tranches.length; i++) require(_startBlockNr > tranches[i].startBlockNr, "Tranche cannot start earlier");

        // Add the tranche to the list
        tranches.push(Tranche(_startBlockNr, _blockCount, _amount));

        // Adjust the balanceTotal
        balanceTotal = balanceTotal.add(_amount);

        // Check if balance and approval are sufficient
        require(token.allowance(wallet, address(this)) >= balanceTotal, "Allowance is too low");
        require(token.balanceOf(wallet) >= balanceTotal, "Balance is too low");

        emit AddTranche(_startBlockNr, _blockCount, _amount);
    }
```

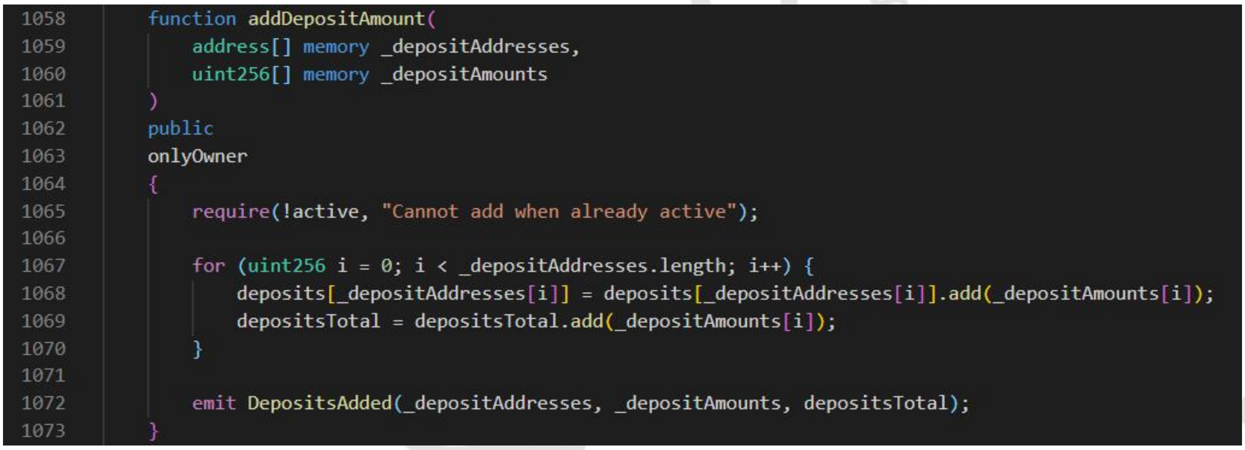
Figure 3 source code of addTranche function

According to the function code, the project party needs to store the tokens on the wallet address and to set the allowance which the wallet address to this contract, and then call this function to add new tranche. After communicating with the project party, the project party pointed out that the wallet address here is a multi-signature address used to manage tokens and meet the project design requirements.

- Related functions: *addTranche*
- Safety Suggestion: None.
- Result: Pass

3.3 Add deposit amount of address

- Description: The contract owner can call the *addDepositAmount* function to add the deposit amount of the specified address.



```
    function addDepositAmount(
        address[] memory _depositAddresses,
        uint256[] memory _depositAmounts
    )
    public
    onlyOwner
    {
        require(!active, "Cannot add when already active");

        for (uint256 i = 0; i < _depositAddresses.length; i++) {
            deposits[_depositAddresses[i]] = deposits[_depositAddresses[i]].add(_depositAmounts[i]);
            depositsTotal = depositsTotal.add(_depositAmounts[i]);
        }

        emit DepositsAdded(_depositAddresses, _depositAmounts, depositsTotal);
    }
```

Figure 4 source code of addDepositAmount function

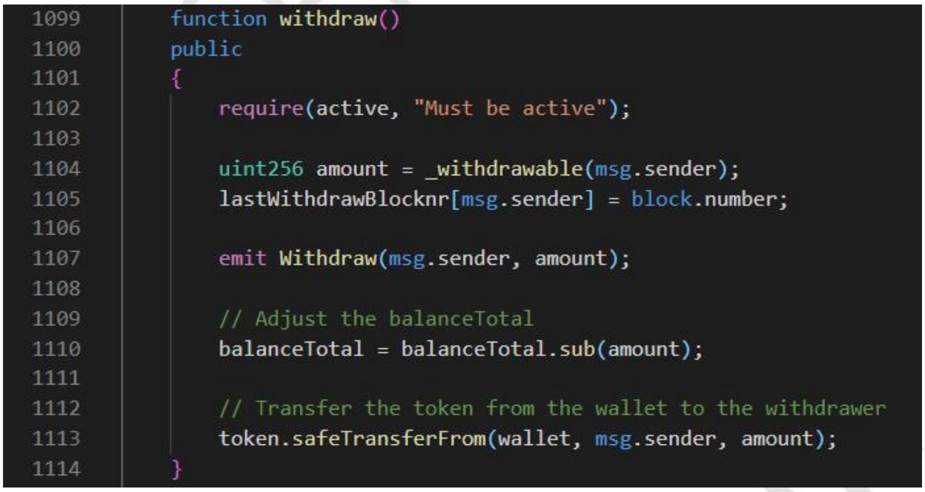
- Related functions: *addDepositAmount*
- Safety Suggestion: None
- Result: Pass

3.4 Contract activation

- Description: After the contract added tranche and set the deposit amount of the specified address, the contract owner can call the *activate* function to activate the contract's withdraw token business.
- Related functions: *activate*
- Safety Suggestion: None
- Result: Pass

3.5 Withdraw token

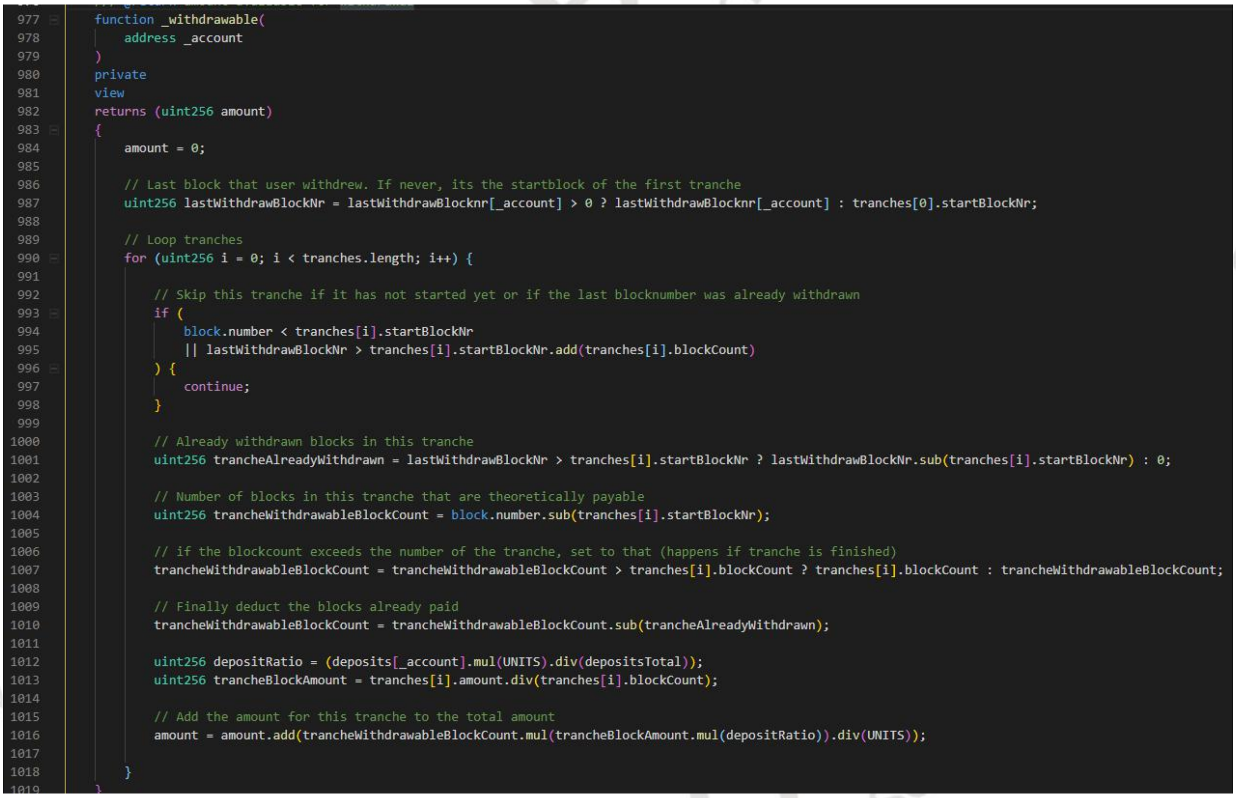
- Description: Users who have deposit tokens can call the *withdraw* function to withdraw tokens. As shown in the figure below, the function requires the contract to be active, and calls the *_withdrawable* function to calculate the amount that the caller can withdraw, and then updates the withdrawal status and sends tokens.



```
    function withdraw()
    public
    {
        require(active, "Must be active");

        uint256 amount = _withdrawable(msg.sender);
        lastWithdrawBlocknr[msg.sender] = block.number;

        emit Withdraw(msg.sender, amount);

        // Adjust the balanceTotal
        balanceTotal = balanceTotal.sub(amount);

        // Transfer the token from the wallet to the withdrawer
        token.safeTransferFrom(wallet, msg.sender, amount);
    }
```

Figure 5 source code of withdraw function

In the *_withdrawable* function, the function needs to accumulate the withdraw-able token amount that the user can withdraw based on the block number of the user's last withdraw token and the block range set by each tranche.



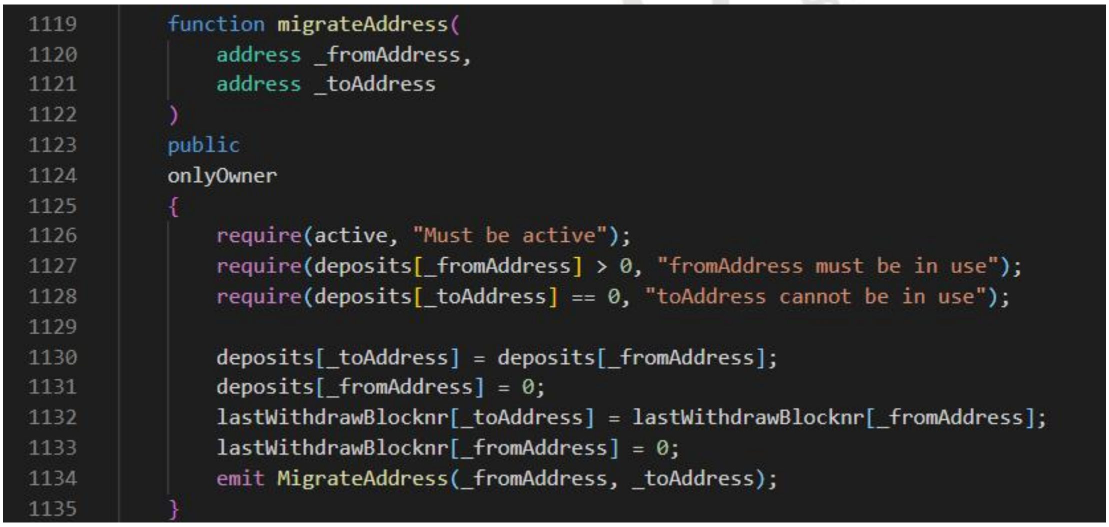
```
function _withdrawable(
    address _account
)
private
view
returns (uint256 amount)
{
    amount = 0;

    // Last block that user withdrew. If never, its the startblock of the first tranche
    uint256 lastWithdrawBlockNr = lastWithdrawBlocknr[_account] > 0 ? lastWithdrawBlocknr[_account] : tranches[0].startBlockNr;

    // Loop tranches
    for (uint256 i = 0; i < tranches.length; i++) {

        // Skip this tranche if it has not started yet or if the last blocknumber was already withdrawn
        if (
            block.number < tranches[i].startBlockNr
            || lastWithdrawBlockNr > tranches[i].startBlockNr.add(tranches[i].blockCount)
        ) {
            continue;
        }

        // Already withdrawn blocks in this tranche
        uint256 trancheAlreadyWithdrawn = lastWithdrawBlockNr > tranches[i].startBlockNr ? lastWithdrawBlockNr.sub(tranches[i].startBlockNr) : 0;

        // Number of blocks in this tranche that are theoretically payable
        uint256 trancheWithdrawableBlockCount = block.number.sub(tranches[i].startBlockNr);

        // if the blockcount exceeds the number of the tranche, set to that (happens if tranche is finished)
        trancheWithdrawableBlockCount = trancheWithdrawableBlockCount > tranches[i].blockCount ? tranches[i].blockCount : trancheWithdrawableBlockCount;

        // Finally deduct the blocks already paid
        trancheWithdrawableBlockCount = trancheWithdrawableBlockCount.sub(trancheAlreadyWithdrawn);

        uint256 depositRatio = (deposits[_account].mul(UNITS).div(depositsTotal));
        uint256 trancheBlockAmount = tranches[i].amount.div(tranches[i].blockCount);

        // Add the amount for this tranche to the total amount
        amount = amount.add(trancheWithdrawableBlockCount.mul(trancheBlockAmount.mul(depositRatio)).div(UNITS));

    }
}
```

Figure 6 source code of _withdrawable function

- Related functions: *withdraw, _withdrawable*
- Safety Suggestion: None
- Result: Pass

3.6 Migrate deposit amount of address

- Description: As shown in the figure below, the contract owner can call the *migrateAddress* function to transfer the deposit amount of the specified address to another address.



```
function migrateAddress(
    address _fromAddress,
    address _toAddress
)
public
onlyOwner
{
    require(active, "Must be active");
    require(deposits[_fromAddress] > 0, "fromAddress must be in use");
    require(deposits[_toAddress] == 0, "toAddress cannot be in use");

    deposits[_toAddress] = deposits[_fromAddress];
    deposits[_fromAddress] = 0;
    lastWithdrawBlocknr[_toAddress] = lastWithdrawBlocknr[_fromAddress];
    lastWithdrawBlocknr[_fromAddress] = 0;
    emit MigrateAddress(_fromAddress, _toAddress);
}
```

Figure 7 source code of migrateAddress function

- Related functions: *migrateAddress*
- Safety Suggestion: None
- Result: Pass

4. Conclusion

Beosin(Chengdu LianAn) conducted a detailed audit on the design and code implementation of the smart contracts project RAMP_PRIVATESALE_VESTING. All the issues found during the audit have been written into this audit report. The overall audit result of the smart contract project RAMP_PRIVATESALE_VESTING is **Pass**.

Beosin

BEOSIN

Blockchain Security

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com