#HashEx
BLOCKCHAIN SECURITY

RSN

smart contracts
final audit report

August 2024

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Penny Picker team to perform an audit of their smart contract. The audit was conducted between 14/08/2024 and 16/08/2024.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at 0xcE35BedB47BBc2e48e635D01354573f9f6E284F3 in the Binance Smart Chain testnet.

Update. The Penny Picker team has responded to this report. The updated contract is available in the Binance Smart Chain testnet at address 0xCf5b7c8CeaB8EE777a4Ba1919a222C537c4C246F and to Binance Smart Chain mainnet at address 0x4F4e82e3de61f9A0a7014c668e1CdA09b557b717.

2.1 Summary

Project name	RSN
URL	https://pennypicker.xyz/
Platform	Binance Smart Chain
Language	Solidity
Centralization level	● Low
Centralization risk	● Low

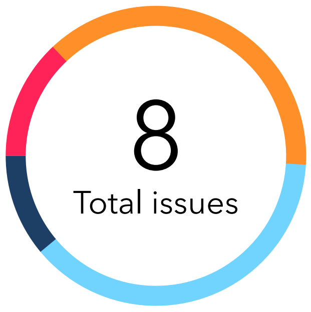
2.2 Contracts

Name	Address
RSNToken	`0x4F4e82e3de61f9A0a7014c668e1CdA09b557b717`

3. Project centralization risks

The contract owners accumulate pre-minted token supply. They can use them to manipulate exchange rates.

4. Found issues



●	Critical	1 (13%)
●	High	3 (38%)
●	Low	3 (38%)
●	Info	1 (11%)

C2a. RSNToken

ID	Severity	Title	Status
C2aI0d	● Critical	Possibility of transfer lock	✓ Resolved
C2aI0f	● High	Allocation amount is not checked	✓ Resolved
C2aI0e	● High	Vesting schedule can be changed	✓ Resolved
C2aI10	● High	Locked amount can exceed balance	✓ Resolved
C2aI13	● Low	Full unlock is not possible	✓ Resolved
C2aI11	● Low	Lack of events	✓ Resolved
C2aI12	● Low	Gas optimizations	✓ Resolved
C2aI14	● Info	Typographical error	✓ Resolved

5. Contracts

C2a. RSNToken

Overview

An ERC20 standard token contract. The initial supply is fixed, i.e. there's no active minting functionality after deployment. The RSN token supports owner-governed allocations - transfers without additional requirements or restrictions to recipient. Users with allocations are allowed to vest funds for arbitrary address.

Issues

C2aI0d Possibility of transfer lock ● Critical ✓ Resolved

Any user with non-zero allocation can vest tokens for an arbitrary address with unlocking schedule of equal-sized chunks with effectively unlimited number of chunks. Setting this length to unreasonably high value prevents any transfer from the address participated in vesting due to block gas limit problem. In other words, any user with positive allocation (which can't be reduced) can block any address from outgoing transfers.

```
/**
 * @dev Set Allocation
 * @param account The address to be allocated
 * @param amount The amount to be allocated
 * @param role The role to be allocated
 */
function setAllocation(
    address account,
  uint256 amount,
    string memory role
) external onlyOwner {
... require(
        _allocations[account].isSet != true,
        "RSN: already registered allocation address"
    );
    _allocations[account] = Allocation(account, amount, role, true);
... }
/**
 * @dev Set Vesting
 * @param account The address to be vested
 * @param amount The amount to be vested
 * @param start The start date(timestamp) to be vested
 * @param timelock The timelock timestamp
 */
function setVesting(
    address account,
    uint256 amount,
    uint256 start,
    uint256[] memory timelock
) public {
    address owner = _msgSender();
    /**
     * @dev Only addresses registered in '_allocation' are permitted to execute.
     */
    require(
        _allocations[owner].isSet == true,
        "RSN: address is not registered in allocation "
);
    _vestings[account] = Vesting(
        account,
        amount,
        start,
        timelock,
        _allocations[owner].role,
        true
);
... }
/**
 * @dev Get the amount of locked tokens
 * @param account address
 * @return lockedToken Amount of locked tokens
 */
function getLockedBalance(address account) public view returns (uint256) {
    ...
    uint256 per = _vestings[account].amount /
        _vestings[account].timelock.length;
    uint256 unlocked = 0;
    for (uint i = 0; i < _vestings[account].timelock.length; i++) {
        if (_vestings[account].timelock[i] < block.timestamp) {
            unlocked += per;
        }
}
    return _vestings[account].amount - unlocked;
}
/**
 * @dev _beforeTokenTransfer
 * @param from from address
 * @param to to address
 * @param amount amount
 */
function _update(
    address from,
address to,
    uint256 amount
) internal override {
    if (from != address(0)) {
        uint256 locked = getLockedBalance(from);
        uint256 accountBalance = balanceOf(from);
        require(
            accountBalance - locked >= amount,
            "RSN: Transfer amount exeeds balance or some amounts are locked."
        );
}
    super._update(from, to, amount);
}
```

Recommendation

Include input validation for the `setVesting()` and `changeVestingDate()` functions.

C2al0f Allocation amount is not checked ● High ✓ Resolved

Allocation is used only for restrictions of vesting creation, but actual allocation amount is never

checked or altered after the initial creation.

```
/**
 * @dev Set Allocation
 * @param account The address to be allocated
 * @param amount The amount to be allocated
 * @param role The role to be allocated
 */
function setAllocation(
    address account,
    uint256 amount,
    string memory role
) external onlyOwner {
    require(account != address(0), "RSN: account from the zero address");
    address owner = _msgSender();
    require(
        balanceOf(owner) >= amount,
        "RSN: allocation amount exceeds balance"
); require(
        _allocations[account].isSet != true,
        "RSN: already registered allocation address"
    );
    _allocations[account] = Allocation(account, amount, role, true);
    _allocationAddresses.push(account);
    transfer(account, amount);
}
/**
 * @dev Set Vesting
 * @param account The address to be vested
 * @param amount The amount to be vested
 * @param start The start date(timestamp) to be vested
 * @param timelock The timelock timestamp
 */
function setVesting(
    address account,
    uint256 amount,
    uint256 start,
    uint256[] memory timelock
) public {
    address owner = _msgSender();
    /**
     * @dev Only addresses registered in '_allocation' are permitted to execute.
     */
    require(
        _allocations[owner].isSet == true,
        "RSN: address is not registered in allocation "
); require(
        balanceOf(owner) >= amount,
        "RSN: vesting amount exceeds balance"
    );
    require(
        _vestings[account].isSet != true,
        "RSN: already registered in vesting"
);
    _vestings[account] = Vesting(
        account,
        amount,
        start,
        timelock,
        _allocations[owner].role,
        true
    );
    _vestingAddresses.push(account);
    transfer(account, amount);
}
```

Recommendation

Consider reducing allocation according to vested amount.

C2al0e Vesting schedule can be changed ● High ✓ Resolved

Any address with non-zero allocation can change vesting date for vested accounts created by another user with possibly different allocation role. Thus, any compromised allocated address can unlock all vested funds at once.

```
/**
 * @dev Change Vesting start date
 * @param account The vested adddress
 * @param start The start date(timestamp) to be vested
 * @param timelock The timelock timestamp
 */
function changeVestingDate(
    address account,
    uint256 start,
    uint256[] memory timelock
) public {
    address owner = _msgSender();
    require(
        _allocations[owner].isSet == true,
        "RSN: address is not registered in allocation "
); require(
        _vestings[account].isSet == true,
        "RSN: account is not registered in vesting"
    );
    _vestings[account].start = start;
    _vestings[account].timelock = timelock;
}
```

Recommendation

Remove the `changeVestingDate()` function or introduce authentication model for it, e.g., allow only calls from the vesting creator.

C2aI10 Locked amount can exceed balance ● High ✓ Resolved

Vesting schedule can be updated by any address with positive allocation. The problem arises if the `changeVestingDate()` function is called after part of vested funds have been unlocked and transferred out. In that case `getLockedBalance()` return value may become greater that balance of user, causing a math underflow error in the `_update()` internal function and in corresponding public transfer functions.

```
/**
 * @dev Change Vesting start date
 * @param account The vested adddress
 * @param start The start date(timestamp) to be vested
 * @param timelock The timelock timestamp
 */
function changeVestingDate(
    address account,
    uint256 start,
    uint256[] memory timelock
) public {
...
    _vestings[account].timelock = timelock;
}
/**
 * @dev Get the amount of locked tokens
 * @param account address
 * @return lockedToken Amount of locked tokens
 */
function getLockedBalance(address account) public view returns (uint256) {
    ...
    for (uint i = 0; i < _vestings[account].timelock.length; i++) {
        if (_vestings[account].timelock[i] < block.timestamp) {
            unlocked += per;
} }
    return _vestings[account].amount - unlocked;
}
/**
 * @dev _beforeTokenTransfer
 * @param from from address
 * @param to to address
 * @param amount amount
 */
function _update(
    address from,
address to,
    uint256 amount
) internal override {
    ...
    uint256 locked = getLockedBalance(from);
    uint256 accountBalance = balanceOf(from);
    require(
        accountBalance - locked >= amount,
        "RSN: Transfer amount exeeds balance or some amounts are locked."
    );
... }
```

Recommendation

Remove the `changeVestingDate()` function or re-implement vesting logic.

C2aI13 Full unlock is not possible ● Low ✓ Resolved

Division error causes small amount of vested funds to remain locked.

```
/**
 * @dev Get the amount of locked tokens
 * @param account address
 * @return lockedToken Amount of locked tokens
 */
function getLockedBalance(address account) public view returns (uint256) {
    ...
    uint256 per = _vestings[account].amount /
        _vestings[account].timelock.length;
... }
```

C2aI11 Lack of events ● Low ✓ Resolved

The functions `setAllocation()`, `setVesting()`, `changeVestingDate()` change important variables in the contract storage, but no event is emitted.

We recommend adding events to these functions to make it easier to track their changes off-chain.

C2aI12 Gas optimizations ● Low ✓ Resolved

1. The Allocation struct contains address of account which is also stored in the `_allocations[]`

mapping as a key and in the `_allocationAddresses[]` array.
2. The Vesting struct contains address of account which is also stored in the `_vestings[]`

mapping as a key and in the `_vestingAddresses[]` array.

3. The Vesting structure contains the start timestamp not used in the contract.

4. Multiple reads from storage in the `getAllocations()` function: `_allocationAddresses.length` variable.

5. Multiple reads from storage in the `getVestings()` function: `_vestingAddresses.length` variable.

6. Multiple reads from storage in the `getLockedBalance()` function: `_vestings[account].timelock.length`, `_vestings[account].amount` variables.

C2aI14 Typographical error

● Info ✓ Resolved

The smart contract code contains typographical errors in error messages and descriptions:

`registerd`, `adddress`, `exeeds` which should be correctly spelled as registered, address, exceeds

to align with standard English grammar conventions.

6. Conclusion

1 critical, 3 high, 3 low severity issues were found during the audit. 1 critical, 3 high, 3 low issues were resolved in the update. The reviewed contracts are highly dependent on the owner's account. See the centralization risks chapter.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

Appendix D. Centralization risks classification

Centralization level

- 🔴 **High.** The project owners can manipulate user's funds, lock user's funds on their will (reversible or irreversible), or maliciously update contracts parameters or bytecode.
- 🟠 **Medium.** The project owners can modify contract's parameters to break some functions of the project contract or contracts, but user's funds remain withdrawable.
- 🟢 **Low.** The contract is trustless or its governance functions are safe against a malicious owner.

Centralization risk

- 🔴 **High.** Lost ownership over the project contract or contracts may result in user's losses. Contract's ownership belongs to EOA or EOAs, and their security model is unknown or out of scope.
- 🟠 **Medium.** Contract's ownership is transferred to a contract with not industry-accepted parameters, or to a contract without an audit. Also includes EOA with a documented security model, which is out of scope.
- 🟢 **Low.** Contract's ownership is transferred to a well-known or audited contract with industry-accepted parameters.

contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx
BLOCKCHAIN SECURITY