Rock'n'Block

SMART CONTRACTS EXPERTISE — RIGHT WAY TO SUCCESS
Binance Smart Chain BOY CZ token report

If you have any questions concerning smart contract design and audit, feel free to contact audit@rocknblock.io

Content

1. Overview

Team of Binance Smart Chain BOY CZ asked us to perform a review of their BEP20 Token contract code. We performed a review of their code from 20.07.2021-23.07.2021, and published this document as a write-up of our findings.

1.1 Terms of Reference for the creation of a smart contract

1.1.1 Token details:

- Token Name - Binance Smart Chain BOY CZ
- Token Symbol - BSCCZ
- Decimals - 8
- Total Supply - 21,000,000 BSCCZ
- Token address - 0xB3b8326C75893632945cA6E4f4Bd76E7a7c7D5Ac
- Token Creator address - 0x3290458d69788302c7dca753896f3c0a10952368
- Token Owner address - 0x10eF43d90cA8BAF45f7a753B931bC064a0AC1EbD
- Type of token - BEP20

1.1.2 Token functions:

transfer	The token contract allows the holder transfer tokens to a specific address
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)
mint	The token contract does not allows the owner or privileged users to mint tokens to a specific address
mintandfreeze	The token contract allows the owner or privileged users to mint and freeze tokens to a specific address with indicate unfreeze date

transferFrom	Allows transfer tokens from the address that previously granted the rights to this operation
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)
burn	The token contract allows to all holders burn their tokens
FreezeTo	The token contract allows to all holders freeze their tokens to any address and specifying unfreeze date
releaseAll	Token owner can unfreeze tokens on all holders address if reached unfreeze date.
releaseOnce	Token holder can unfreeze tokens on his address if reached un-freeze date
pause	The token contract allows the owner pause the token transfers and other operations
unpause	The token contract allows the owner unpause the token transfers and other operations
renounceOwnership	The token contract allows the owner to renounce ownership. After calling renounceOwnership no one can be owner of the contract
transferOwnership	The token contract allows the owner transfer ownership to anoth-er address
approve	Allows token holders to transfer the right to manage the token on your balance to a third-party address
increaseApproval	Allows token holders to increase the amount of tokens, the right to control which is transferred to a third-party address
decreaseApproval	Allows token holders to decrease the amount of tokens, the right to control which is transferred to a third-party address

2. Introduction

2.1. Authenticity

The contracts audited are a subset of the contracts compiled and deployed in the blockchain.

https://bscscan.com/address/0xB3b8326C75893632945cA6E4f4Bd76E7a7c7D5Ac#code

The assessed BSCCZ token smart contract components were written in Solidity, and the version used for this report is commit:

https://github.com/Rock-n-Block/AUDIT/blob/main/BSCCZ

https://github.com/Rock-n-Block/AUDIT/commit/416ef6acb898f3188a41294755280a9e07017bda

2.2. Scope

The audit reviewed contract source code from Bscscan. Contract were reviewed in the context of the flattened file, which included a single solidity file. The review performed did not assess any scripts, tests, or other non-Solidity files.

2.3. Methodology

This audit was performed as a comprehensive review of the codebase and takes into consideration both the Solidity code, as well as the target platform: Binance Smart Chain network. The Solidity was reviewed not just for common vulnerabilities and antipatterns, but also for its parity with the intent of the deployer, for its efficiency, and for the practices used during development.

2.4. Description of the complex of procedures for reviewing the smart contract

2.4.1 Primary architecture review

- Checking the architecture of the contract.
- The correctness of the code.
- Check for linearity, shortness, and self-documentation.
- Static verification and code analysis for validity and the presence of syntactic errors.

2.4.2 Comparison of requirements and implementation

- Checking the code of the smart contract for compliance with the requirements of the customer code logic, writing algorithms, matching the initial constant values.
- Identification of potential vulnerabilities

2.4.3 Testing according to the requirements

- Control testing of the smart contract for compliance with specified customer requirements.
- Running properties tests of the smart contract in the test net.

2.5. Risk Assessment

Findings were categorized using a risk rating model based on the OWASP method. Each vulnerability takes into consideration the impact and likelihood of exploitation, as well as the relative ease with which the vulnerability is resolved; findings that permeate throughout the codebase will require much more review and work to solve and are rated higher as a result.

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology:

• Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

• Impact measures the technical loss and business damage of a successful attack;

• Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: H, M and L, i.e., high, medium and low respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., Critical, High, Medium, Low shown in following Table

Table: Vulnerability Severity Classification

impact				
	High	**Critical**	**High**	**Medium**
	Medium	**High**	**Medium**	**Low**
	Low	**Medium**	**Low**	**Low**
		High	*Medium*	*Low*
		Likelihood		

2.6. Disclaimer

This document reflects the understanding of security flaws and vulnerabilities as they are known to Rock`n`Block, and as they relate to the reviewed project. This document makes no statements on the viability of the project or the safety of its code. This audit does not represent investment advice and should not be interpreted as such.

3. Findings

3.1. Critical Severity

No critical-severity vulnerabilities were found.

3.2. High Severity

No high-severity vulnerabilities were found.

3.3. Medium Severity

No medium-severity vulnerabilities were found.

3.4. Low Severity

Disparity of expectation in release functions: Users use releaseOnce() and releaseAll() to release their frozen tokens once the freeze period has elapsed. In the event a user does not hold any frozen tokens eligible for release, the releaseOnce() function reverts state changes. This is not the case for releaseAll(), which will simply do nothing. While this does not pose a significant danger for users, we recommend the inconsistency be addressed.

Overuse of public function visibility: The reviewed token contract is assembled using a script which generates a file of constants with which the token contract will set its initial values. Because each constant is marked public, Solidity implicitly creates a publicly visible getter function with the same name. While using constants is generally efficient, excessive use of public fields:

1. Makes a contract more expensive to deploy (longer bytecode)

2. Makes a contract more expensive to use, as each additional function selector created by these implicit getters means more options to traverse at runtime.

Consider removing the word public from each constant unless absolutely necessary. They will be set to the default, internal, meaning they will still be accessible internally to the contract.

3.5. Notes and Recommendations

Important - improper input sanitization during key generation, and mixing of user frozen token records:

The reviewed token contract inherits from FreezableToken.sol, an extension of the BEP20 standard implementing token transfers that can transfer time-locked tokens.

Users are able to use the function freezeTo(address, uint, uint64) to transfer tokens to an address which cannot be transferred again until the specified time period has elapsed.

Frozen tokens are saved in order of release date, for more efficient access; a user must invoke releaseOnce() or releaseAll() to release their tokens once the period has elapsed. Release dates are linked together through the generation of a unique key using an internal key generator, toKey(address, uint). As the input parameters suggest, the key is generated using the frozen token holder's address, and the release date. The purpose of this is to ensure that if frozen tokens are sent to a user multiple times with the same release date, the keys generated will be the same each time.

Each generated key is linked to the next sequential release date of a user's frozen tokens in the chains mapping. The actual amount frozen for each release date is located in the freezings mapping. Both take as input the generated key for a release date. chains returns the next date in the sequence, while freezings returns the number of tokens frozen at the given release date. To clarify: when a user is sent frozen tokens, a key is generated in toKey using their address and the release date. The number of frozen tokens is recorded under this key in freezings. If the user has tokens frozen at a later release date than this latest batch, the chains mapping will point to the next release date in the sequence, from which a new key can be calculated using toKey, which will point to its own amount of frozen tokens and next sequential date if it exists. The pattern repeats until chains[key] returns 0, at which point the contract knows it has arrived at the end of the user's frozen token sequence.

The security of each user's frozen token holdings relies on each key generated as being unique. If this were not the case, an attacker could craft an address and release date combination whose generated key matched the key created by a different user to store a record of their frozen tokens, allowing them to lay claim to tokens held frozen by other people.

The key generation used is the crux of the problem: the key generated is 32 bytes in size, the default size used by Solidity for most values. The components of the key used, if the key is to remain unique, should add up to 32 bytes in size. On the surface, this appears to be the case: an initial mask (4 bytes) is followed by the holder's address (20 7 bytes), followed by the release date, which is assumed to be 8 bytes in size, completing the 32-byte key.

The toKey function, however, only assumes, but does not require, the provided release date to be 8 bytes in size. Release dates can be given to the function as a default 32-byte value, as toKey does not check the size. In the case an attacker is able to provide the function with a 32-byte value, the key generation method used no longer produces unique keys. Instead, the additional bytes overlap with the token holder's address, allowing the attacker to generate release dates that act to completely equate their address with any other address when fed through the key generator. The resulting much later date chosen should impose a waiting period on the attacker, but the release functions only compare the last 8 bytes of the release time with the current time. This means an attacker would be able to access the rightful owner's frozen tokens as soon as they could, for any address and any release date.

Improper input sanitization in toKey means that a developer writing or using these contracts needs to remember, every time they use the key generator, to only pass 8 bytes of information into the key generator. In the reviewed code, the input size was correctly altered from its default "32" to the secure "8" a total of 12 out of 12 times. Given that 32-bytes is the default size, this presents a classic anti-pattern - requiring that input sanitization (validating the input size) be performed external to the critical function, rather than simply ensuring the critical function validates the input itself.

Compounding this risk is the structure of the chains mapping, which only maps from keys to release dates, meaning that multiple users essentially "share" the same mapping. As long as keys are unique, this does not pose a significant risk. However, if an attacker is able to find values for which collisions are created, users sharing the same mapping means the attacker is able to affect the holdings of a much larger proportion of frozen token holders.

Our recommendation is as follows:

1. Do not rely on simple arithmetic operations to generate a unique key. Instead, keys should be generated using keccak256, a secure hashing function.

2. Change the chains mapping to use separate address spaces for each user.

3. Instead of a mapping from bytes32 => uint64, the mapping should map from address => bytes32 => uint64. A similar addition should be made to the freezings mapping.

4. If the codebase relies on a key generator function like toKey, it should check that the input parameters to key generation match the sizes it expects for safe key generation. For example, if dates are only 8 bytes in length the input parameter should read uint64, not uint; the latter defaults to uint256, which uses 32 bytes.

This issue was included as a note, as it does not provide an angle of attack in the reviewed contracts: key generation is successfully handled in the reviewed code. However, we strongly recommend considering and incorporating these changes prior to release and use, and especially if source code of this contract are intended to be used for future deployments.

Redundant modifier use in MintableToken: The modifier hasMintPermission is logically equivalent to the onlyOwner permission. Consider removing and using onlyOwner in all cases.

Unnecessary boolean return from public functions: In MintableToken.mint, MintableToken.finishMinting, and FreezableMintableToken.mintAndFreeze, a boolean is returned from the public function. However, logic in these functions dictates that if execution should fail for any reason (insufficient permissions, invalid contract state, etc), then each function will simply revert all state changes. As such, the boolean return value serves to only ever return true; false is never returned. (As an aside, the boolean return values from all BEP20 functions must still be included. While they are just as redundant as their use in this function, the BEP20 standard requires they be included. However, no such standard requires that mint, or any other listed functions, returns true.

4. Manual testing

4. Testings

4.1. **Successful** Deployment token in test net. Open link

4.2. **Successful** Check name, symbol, decimals.

4.3. **Successful** Owner of contracts sets correctly.

4.4. **Successful** Checking the distribution function of tokens. Tokens are distributed correctly. Open link

4.5. **Successful** Transfer tokens from address to address. Open link

4.6. **Successful** The MINT function. Tokens are minted and sent to an address. Open link

4.7. **Successful** The MintAndFreeze function. Tokens are minted, frozen and sent to an address. Tokens are displayed on the address, but cannot be sent to another address until they are unfrozen by the token holder after the unfreezing date. Open link

4.8. **Successful Burn** Burning tokens from management address. Open link

4.9. **Successful** Finalize. The mintable function is disabled. No more tokens can be minted. Open link

4.10. **Successful** transferOwnership. Transfer of rights to manage the token. Open link

5. Documents and Resources

5.1. Source

The used source code can be found in Bscscan: https://bscscan.com/address/0xB3b8326C75893632945cA6E4f4Bd76E7a7c7D5Ac#code

The original source code used can be found in the Rock`n`Block repository: https://github.com/Rock-n-Block/AUDIT/blob/main/BSCCZ

https://github.com/Rock-n-Block/AUDIT/commit/416ef6acb898f3188a41294755280a9e07017bda

5.2. References

OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

6. Conclusion

The information in this review is a list of recommendations on what needs to be done to ensure the quality and security of the smart contract. The Rock`n`block experts conducted the verification of the smart contract. Based on the results of the reviewing and testing, it is established that the token smart contract complies with the specifications specified in the terms of reference.

During the reviewing and testing of the contracts, critical errors and possible vulnerabilities were not detected. Outside of the included notes, the code reviewed was simple and clean. The formatting, naming, and other conventions used were fairly regular, and the inheritance structure was well-organized, resulting in a codebase that was easier to review.

For all questions regarding the review and testing of the smart contract, we recommend contacting audit@rocknblock.io