securing

www.securing.biz

Smart Contracts Security Review of BasketCoin Token for BSKT Technologies Limited

Report

document version: 1.0

author: Damian Rusinek

test time period: 2021-04-14 – 2021-04-15

report date: 2021-04-15

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

1.1. Testing overview

BasketCoin Token (BSKT) is a token designed in the Ethereum ecosystem (ERC20) and its basic characteristics can be briefly presented using the following wording:

- deflationary,
- with the system of repurchase and burning of tokens,
- with collateral – covering its value,
- with options for staking,
- with fair, transparent and well-constructed tokenomics,
- with the subsequent implementation of management tokens.

The security review of the BasketCoin Token smart contract was meant to verify whether the proper security mechanisms were in place to prevent users from abusing the contracts' business logic and to detect the vulnerabilities which could cause financial losses to the client or its customers.

Security tests were performed using the following methods:

- Source code review,
- Verification of compliance with Smart Contracts Security Verification Standard (SCSVS) in a form of code review.

1.2. Summary of test results

- During the penetration testing neither critical vulnerabilities (vulnerabilities which are easy to exploit and have high risk impact) nor medium risk vulnerabilities were found.
- There were 3 vulnerabilities with low impact on risk identified. Their exploitation results are following:
    - Burning more tokens than allowed according to the documentation.
    - Lost control over the token and potential loss of all Ethers on the contract if it happens during the sale.
    - Transferring victim's tokens on behalf of the previous allowance as well as the new allowance.
- Two recommendations have been proposed that do not have any direct risk impact. However, it is suggested to implement them due to good security practices.
- The scope of the test included the source code of the token contract already deployed on mainnet. Security review of the development process (including testing) was out of scope.

1.3. Disclaimer

The security review described in this report is not a security warranty. It does not guarantee the security or correctness of reviewed smart contracts. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best security practices, security reviews and formal verification, ending with constant monitoring and incident response. Therefore, we strongly recommend implementing security mechanisms at all stages of development and maintenance.

2. SUMMARY OF IDENTIFIED VULNERABILITIES

2.1. Risk classification

Vulnerabilities	
Risk impact	**Description**
Critical	Vulnerabilities that affect the security of the entire system, all users or can lead to a significant financial loss (e.g. tokens). It is recommended to take immediate mitigating actions or limit the possibility of vulnerability exploitation.
High	Vulnerabilities that can lead to escalation of privileges, a denial of service or significant business logic abuse. It is recommended to take mitigating actions as soon as possible.
Medium	Vulnerabilities that affect the security of more than one user or the system, but might require specific privileges, or custom prerequisites. The mitigating actions should be taken after eliminating the vulnerabilities with critical and high risk impact.
Low	Vulnerabilities that affect the security of individual users or require strong custom prerequisites. The mitigating actions should be taken after eliminating the vulnerabilities with critical, high, and medium risk impact.
Recommendations	
Methods of increasing the security of the system by implementing good security practices or eliminating weaknesses. No direct risk impact has been identified. The decision whether to take mitigating actions should be made by the client.	

2.2. Identified vulnerabilities

Vulnerability	Risk impact
4.1 Burning tokens below the specified minimum total supply	**Low**
4.2 One-step ownership transfer	**Low**
4.3 Front-runnable approve function	**Low**
Recommendations	
5.1 Use simple to understand variable names	
5.2 Validate the fees	

3. PROJECT DESCRIPTION

3.1. Basic information

Testing team	Damian Rusinek Paweł Kuryłowicz
Testing time period	2021-04-14 – 2021-04-15
Report date	2021-04-15
Version	1.0

3.2. Target in scope

The object being analysed was BasketCoin token contract accessible in the following GitHub repository:

- https://github.com/BasketCoinBSKT/BSKTCoin
  (commit: 6ed5704f181f8f2d56768e7631c01a69b3c36476)

Contracts in scope:

- BasketCoin (token.sol)

3.3. Threat analysis

The key threats were identified as follows:

1. Ether theft,
2. BSKT theft,
3. Lack of compliance with whitepaper,
4. Purchasing more tokens that available.

3.4. Scope

Following the specification, the tests covered:

- Source code review,
- Static source code analysis,
- Dynamic smart contract analysis (automated tools),
- Compliance with Smart Contracts Security Verification Standard (SCSVS), version 1.1.

3.5. Exclusion and remarks

The scope of the test included the source code of the token contract already deployed on mainnet. Security review of the development process (including testing) was out of scope.

4. VULNERABILITIES

4.1. Burning tokens below the specified minimum total supply

Risk impact	Low
Exploitation conditions	Access to the tokens and owner account.
Exploitation results	Burning more tokens than allowed according to the white paper.
Remediation	Add the following requirement in the *burn* function: `require(totalSupply().sub(amount) >= minTotalSupply, "BSKT: cannot burn below the limit");` *The variable name is changed according to the recommendation 5.1.*

Vulnerability description:
The white paper says:
*"In order to meet inflation, the initial supply of BSKT tokens was set at 21 million (21,000,000). BSKT is programmed to stop firing when its quantity reaches 10% of the maximum supply. **This means***
no less, no more, that only 2.1 million (2,100,000) BSKT will eventually remain in circulation."
However, the *burn* function allows to burn more tokens and make the total supply lower than 2.1 million.
The risk impact has been lowered as this can be exploited by the owner only and they cannot burn other's tokens.

Test case:
Below is the *burn* function implementation which does not control the amount that can be burnt.

```
function burn(uint256 amount) external onlyOwner {
    super._burn(_msgSender(), amount);
}
```

4.2. One-step ownership transfer

Risk impact	Low
Exploitation conditions	Mistaken transaction that sets a new owner to invalid address.
Exploitation results	Lost control over the token and potential loss of all Ethers on the contract if it happens during the sale.
Remediation	Make the ownership transfer a 2 step process where the pending new owner has to claim the ownership and the current owner can change the pending owner before the claim.

Vulnerability description:

The *transferOwnership* function from *Ownable* contract sets the new owner immediately. If the new owner address is incorrect, the contract loses the owner and no one can control it.

Here is an example of code that makes this process a 2-step process where the new owner must confirm the ownership transfer.

```
address private _pendingOwner;

modifier onlyPendingOwner() {
if (msg.sender == _pendingOwner)
    _;
}

function transferOwnership(address newOwner) public override onlyOwner {
_pendingOwner = newOwner;
}

function claimOwnership() onlyPendingOwner {
_owner = _pendingOwner;
_pendingOwner = 0x0;
}
```

4.3. Front-runnable approve function

Risk impact	Low
Exploitation conditions	Front-running victim's transaction that changes existing victim's allowance for attacker as spender (by using *approve* function).
Exploitation results	Transferring victim's tokens on behalf of the previous allowance as well as the new allowance.
Remediation	As there is no general remediation for this vulnerability, we recommend to allow to either set allowance to 0 (**from** any value) or from 0 (**to** any value). Also, do not use the *approve* function in a DApp. Use the *increaseAllowance* and *decreaseAllowance* instead.

Vulnerability description:
The BasketCoin contract implements ERC20 standard which contains *approve* function. The function allows to set a given allowance of sender's tokens to a defined spender.
Front-running the transaction that changes existing allowance makes it possible for the malicious spender to transfer victim's tokens before the *approve* function call is added to block and again, using another transaction, to transfer the newly set amount of tokens. As the consequence, the attacker has transferred the sum of both allowances and not the second allowance only, as it was desired by the victim.
This vulnerability is common vulnerability for ERC20 implementations and there is no general mitigation. The recommendation is to allow either to set allowance to 0 (from any value) or from 0 (to any value).

Test case:
The vulnerable implementation of *approve* function in *BasketCoin* contract can be found below:

```
function _approve(address owner, address spender, uint256 amount) internal
virtual {
     require(owner != address(0), "ERC20: approve from the zero address");
     require(spender != address(0), "ERC20: approve to the zero address");

     _allowances[owner][spender] = amount;
     emit Approval(owner, spender, amount);
}
```

Consider the following calls:

```
Victim submits: approve(Spender, 1000) – ADDED TO MEMPOOL
Transaction validated: Victim: approve(Spender, 1000)

# Victim wants to set allowance to 100 because 1000 was a mistake
Victim submits: approve(Spender, 100) – ADDED TO MEMPOOL

# Spender becomes MaliciousSpender
```

```
MaliciousSpender submits: transferFrom(Victim, 1000) – ADDED TO MEMPOOL
Transaction validated: MaliciousSpender: transferFrom(Victim, 1000)
Transaction validated: Victim: approve(Spender, 100)
MaliciousSpender submits: transferFrom(Victim, 100) – ADDED TO MEMPOOL
Transaction validated: MaliciousSpender: transferFrom(Victim, 100)
```

This scenario presents a chain of transactions where the malicious spender was able to transfer 1100 tokens, while the maximum allowance being set by the victim was 1000.

5. RECOMMENDATIONS

5.1. Use simple to understand variable names

Description:
The variable *minTotalSupplyToBurn* is not self-explanatory and it is hard to guess (without looking at the source code) that it specifies the minimum total supply.

How to implement:
Change variable name, e.g. to *minimumTotalSupply*.

5.2. Validate the fees

Description:
Currently the fees are calculated using the following code:

```
uint256 burnAmount = amount.mul(burnFee).div(100);
uint256 stakeAmount = amount.mul(stakeFee).div(100);
```

The amount lower than 100 allows to bypass the fees as they would be equal to 0. It is recommended to check whether the fees are higher than 0.
This has been reported as recommendation, because the low amount is worth less than the transaction cost.

How to implement:
Check whether the fees are higher than 0.
The other approach is to calculate the amount to be transferred and then subtract it from the original amount to get the fees. In this case, the amount would be zeroed for small values, not the fees.

6. TERMINOLOGY

This section explains the terms that are related to the methodology used in this report.

Threat

Any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service.[1]

Vulnerability

Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.[1]

Risk

The level of impact on organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals resulting from the operation of an information system given the potential impact of a threat and the likelihood of that threat occurring.[1]

The risk impact can be estimated based on the complexity of exploitation conditions (representing the likelihood) and the severity of exploitation results.

		Complexity of exploitation conditions		
		Simple	Moderate	Complex
Severity of exploitation results	Major	Critical	High	Medium
	Moderate	High	Medium	Low
	Minor	Medium	Low	Low

[1] NIST FIPS PUB 200: Minimum Security Requirements for Federal Information and Information Systems. Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology.

7. CONTACT

Person responsible for providing explanations:

Damian Rusinek
e-mail: Damian.Rusinek@securing.pl
tel.: +48 12 425 25 75

securing
http://www.securing.pl
e-mail: info@securing.pl
Kalwaryjska 65/6
30-504 Kraków
tel./fax.: +48 (12) 425 25