SMART CONTRACT AUDIT REPORT

for

BP Token

Prepared By: Yiqun Chen

PeckShield

Aug 09, 2021

Document Properties

Client	BunnyPark
Title	Smart Contract Audit Report
Target	BP Token
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	Aug 09, 2021	Jing Wang	Final Release
1.0-rc	Aug 07, 2021	Jing Wang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the **BP Token** contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract can be further improved due to the presence of certain issues related to ERC20-compliance, security, or performance. This document outlines our audit results.

1.1 About BP Token

BP Token is an ERC20-compliant token that is closely related to the `BunnyPark` protocol in minting reward tokens. The main functionality includes full ERC20 compatibility with additional extensions that are designed to interact with `BunnyPark`'s framing protocol for reward collection. The basic information of BP Token is as follows:

Table 1.1: Basic Information of BP Token

Item	Description
Issuer	BunnyPark
Website	https://www.bunnypark.com
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	Aug 09, 2021

In the following, we show the etherscan link for the `BP` token contract used in this audit. Note this token contract assumes a trustworthy `timelock` account that is privileged to mint additional tokens into circulation.

- https://bscscan.com/address/0xacb8f52dc63bb752a51186d1c55868adbffee9c1

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/Jackluren/BunnyPark-BP.git (bdf0ff4)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- 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 Table 1.2.

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

We perform the audit according to the following procedures:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead of Transfer
	Costly Loop
	(Unsafe) Use of Untrusted Libraries
	(Unsafe) Use of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section 3)
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `BP Token` contract. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	0	
Informational	2	■ ■
Total	3	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC20 compliance issue was found, and our detailed checklist can be found in Section 3. However, the smart contract implementation can be improved because of the existence of 1 medium-severity vulnerability and 2 informational recommendations.

Table 2.1: Key BP Token Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue Of Admin Roles	Security Features	Mitigated
PVE-002	Informational	Constant/Immutable States If Fixed Or Set at Constructor()	Coding Practices	Fixed
PVE-003	Informational	Lack of Emitting Meaningful Events	Coding Practices	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for our detailed compliance checks and Section 4 for elaboration of reported issues.

3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited BP Token. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

4 Detailed Results

4.1 Trust Issue Of Admin Roles

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `BPToken`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `BPToken` token contract, there is a privileged `owner` account (assigned in the `constructor`) that plays a critical role in governing and regulating the token-related operations (e.g., mints/burns tokens).

To elaborate, we show below the `mint()` function in the `BPToken` contract. This function allows the `owner` to mint infinite amount of `BPs` to himself.

```
function mint(uint256 amount) public onlyOwner returns (bool) {
  _mint(_msgSender(), amount);
  return true;
}
```

Listing 4.1: `BPToken::mint()`

What's more, if we examine the the `mintTo()/burn()` routine in the `BPToken` contract. These routines allow the `owner/keeper` to mint/burn any amount of `BPs` to any account. Note that the privileged group `keeper` can also be updated by `owner` via the functions `setKeeper()/removeKeeper()`.

```
function mintTo(address to, uint256 amount) override public ownerOrKeeper(msg.sender)
    returns (bool) {
  _mint(to, amount);
  return true;
}

function burn(address account, uint256 amount) override public ownerOrKeeper(msg.
    sender) returns (bool) {
```

```
        _burn(account, amount);
        return true;
    }
```

Listing 4.2: `BPToken::mintTo()/burn()`

```
    function setKeeper(address addr) public onlyOwner {
      keeperMap[addr] = true;
    }

    function removeKeeper(address addr) public onlyOwner {
        keeperMap[addr] = false;
    }
```

Listing 4.3: `BPToken::setKeeper()/removeKeeper()`

We understand the need of the privileged functions for contract upgrade, but at the same time the extra power to the admin roles may also be a counter-party risk to the contract users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance. Also list `keeper` accounts granted by `owner` explicitly to users.

Status This issue has been confirmed by the teams. And the team transferred the privileged account to the `Timelock` contract (0x88b048191b071ed1bcd1ff7c7c21a697ec86811c). Also, they provide the list of keeper accounts in the `KeeperMap`, which are shown in Table 4.1.

Table 4.1: Current Keeper Accounts of `keeperMap` (as of 2021/08/03)

Contract	Address
presale classic sharedCARD harvest mine	0x0288dda09e7d2e68edce896de4a045c1f8176fee
classic NFT mine	0x03a40aba6865ba5045d80137dedf46fb3312a4ee
farm	0x13ddd28adefc0ace82b06673407a2aca0fc70cd8
BP->BP pool	0x3b0969c3f03bc0ab35ff9b8784904de6c381250a
classicNFT mine	0x5135e2f58d5d8fb85019990fee72951b63ac6524
CAKE–> BP Pool	0x713ddddabb134f5ab50b090d8d1888f4837e9632
classic –> shop	0x8ae187c2877edf2c2d510658ef9037d10f23977d
bigbang NFT mine	0x9b120ce24e9715b7ba1fb9ed59d399cf2f8341e3
dreamNftAttachAddr	0xd991537678236b9b1cbb8380cf6fdab944c61d3a
collection mine	0xf3a8943fbe20c24fcc278df6504ef6de6830ca71

4.2 Constant/Immutable States If Fixed Or Set at Constructor()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BPToken`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variables defined in `BPToken`. If there is no need to dynamically update these key state variables, e.g., `_name` and `_symbol`, they can be declared as `immutable` for gas efficiency.

In addition, we notice the state variable `_decimals` is a constant and we can simply define it as a `constant` to avoid gas cost for the access.

```
    contract BPToken is Context, IBPToken, Ownable {
        ...
        string private _name;
        string private _symbol;
        uint8 private _decimals;
        ..
```

Listing 4.4: `BPToken.sol`

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status This issue has been addressed in the following commit: `bdf0ff4`.

4.3 Lack of Emitting Meaningful Events

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: BPToken
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [3]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the BPToken contract as an example. While examining the events that reflect the BPToken dynamics, we notice there is a lack of emitting related events that reflect important state changes. Specifically, when the keeperMap is being changed, there is no respective event being emitted to reflect the adding/removing of keeper (line 49).

```
    function setKeeper(address addr) public onlyOwner {
        keeperMap[addr] = true;
    }
    function removeKeeper(address addr) public onlyOwner {
        keeperMap[addr] = false;
    }
```

Listing 4.5: BPToken::setKeeper()/removeKeeper()

Recommendation Properly emit the related setKeeper/removeKeeper event when the keeperMap is being updated.

Status This issue has been addressed in the following commit: 220b659.

5 Conclusion

In this security audit, we have examined the design and implementation of the `BP Token` contract. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, although no critical or high level vulnerabilities were discovered, we identified three issues that were promptly confirmed and addressed by the team. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

[7] PeckShield. PeckShield Inc. https://www.peckshield.com.