



SMART CONTRACT AUDIT REPORT

for

FBS\$ Token



Prepared By: Xiaomi Huang

PeckShield

June 27, 2025

Document Properties

Client	FBS\$ Token
Title	Smart Contract Audit Report
Target	FBS\$ Token
Version	1.0
Author	Patrick Lou
Auditors	Patrick Lou, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	June 27, 2025	Patrick Lou	Final Release

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About FBS\$ Token	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	8
2.1	Summary	8
2.2	Key Findings	9
3	ERC20 Compliance Checks	10
4	Detailed Results	13
4.1	Revisited Logic of addNewYearForInflationaryModel()	13
4.2	Trust Issue of Admin Keys	14
5	Conclusion	17
	References	18

1 | Introduction

Given the opportunity to review the design document and related source code of the FBS\$ 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 FBS\$ Token

FBS\$ Token is an ERC20 compliant token which is mintable, burnable and pausable. It includes a 10-year period specification of the inflationary model per year with a max allowable tokens to mint within a year. The max supply of the token can be changed after 10 years. The basic information of the audited FBS\$ contract is as follows:

Table 1.1: Basic Information of FBS\$ Token

Item	Description
Name	FBS\$ Token
Type	ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	June 27, 2025

1.2 About PeckShield

PeckShield Inc. [6] 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 [5]:

- 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	High	Medium	Low
	Critical	High	Medium
	High	Medium	Low
	Medium	Low	Low
Likelihood			
High Medium 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 FBS\$ 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	1	
Informational	0	
Total	2	

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 any identified issue(s) 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 of each of them 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 low-severity vulnerability and 1 informational suggestion.

Table 2.1: Key FBS\$ Token Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited Logic of addNewYearForInflationaryModel()	Coding Practices	Fixed
PVE-002	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 an ERC20 inconsistency or incompatibility issue found in the audited token contracts. Specifically, zero amount transfers are not allowed and related events are not fired. In the surrounding two tables, we outline the respective list of basic `view-only` functions

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 <i>address(0x0)</i> when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

(Table 3.1) and key state-changing functions (Table 3.2) according to the widely-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 Revisited Logic of addNewYearForInflationaryModel()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: FBS\$
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

The FBS\$ token includes a 10-year period specification of the inflationary model per year with a max allowable tokens to mint within a year. The max supply of the token can be changed after 10 years and the admin can set the inflationary model for each year with the `addNewYearForInflationaryModel()` routine. While examining its logic, we observe the logic need to be improved. To illustrate, we show below the `addNewYearForInflationaryModel()` routine.

The `inflationaryModelTotalYears` is initialized to 10 in the constructor and used as the key for the `inflationaryModelPerYear` mapping state variable. It will be set to 11 (line 137, 139) when this routine is first called after 10 years which indicates the inflation model data for 11th year is stored under key 11. However, this may not be true if this routine does not get called in 11th year. In particular, the `inflationaryModelTotalYears` may still be 11 when it was first called in subsequent years that come after year 11. The key should reference the current year according to the design and in this case, the data stored will be inconsistent with the current year, resulting in an incorrect calculation.

```
121 function addNewYearForInflationaryModel(uint256 maxMintAmount) external {
122     uint256 currentYear = getCurrentYear();
123     require(
124         currentYear > 10,
125         "FBS$: cannot add new inflationary model before 10-year period"
126     );
127     require(
128         hasRole(DEFAULT_ADMIN_ROLE, _msgSender()),
129         "FBS$: must have admin role to set the inflationary model"
```

```

130     );
131     require(maxMintAmount > 0, "FBS$: maxMintAmount must be positive");
132     require(
133         (ERC20.totalSupply() + (maxMintAmount * E18)) <= MAX_SUPPLY,
134         "FBS$: maxMintAmount invalid as max supply exceeded"
135     );
136
137     inflationaryModelTotalYears++;
138
139     inflationaryModelPerYear[inflationaryModelTotalYears] = YearlyMintInfo(
140         maxMintAmount,
141         0,
142         maxMintAmount
143     );
144
145     emit EvtAddNewYearForInflationaryModel(
146         inflationaryModelTotalYears,
147         maxMintAmount
148     );
149 }

```

Listing 4.1: FBS\$::addNewYearForInflationaryModel()

Recommendation Revisit the logic for `addNewYearForInflationaryModel()` to handle this case properly.

Status This issue has been fixed in this commit: [4aaa44b](#).

4.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: FBS\$
- Category: Security Features [\[3\]](#)
- CWE subcategory: CWE-287 [\[2\]](#)

Description

In the FBS\$ token protocol, there are special administrative accounts, i.e., `DEFAULT_ADMIN`, `MINTER` and `PAUSER`. Those privileged accounts play critical roles in governing and regulating the protocol-wide operations (e.g., system parameters configuration, mint new tokens, pause the protocol). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged accounts need to be scrutinized. In the following, we examine the privileged `DEFAULT_ADMIN_ROLE` account and its related privileged accesses in current contract.

To elaborate, we show one of the related routine from the FBS\$ token contract. The routine allows the DEFAULT_ADMIN_ROLE account to set the inflationary model (line 86-89).

```

82     function updateInflationaryModelPerYear(
83         uint256 yearID,
84         uint256 maxMintAmount
85     ) external {
86         require(
87             hasRole(DEFAULT_ADMIN_ROLE, _msgSender()),
88             "FBS$: must have admin role to set the inflationary model"
89         );
90         require(yearID > 0, "FBS$: yearID must be positive");
91         uint256 currentYear = getCurrentYear();
92         require(
93             yearID >= currentYear,
94             "FBS$: yearID must be from current year afterwards"
95         );
96
97         require(maxMintAmount > 0, "FBS$: maxMintAmount must be positive");
98         require(
99             (ERC20.totalSupply() + (maxMintAmount * E18)) <= MAX_SUPPLY,
100             "FBS$: maxMintAmount invalid as max supply exceeded"
101         );
102
103         require(
104             inflationaryModelPerYear[yearID].maxMintAmount > 0,
105             "FBS$: inflationary model for the given year not exist"
106         );
107
108         require(
109             maxMintAmount >= inflationaryModelPerYear[yearID].currentMintAmount,
110             "FBS$: maxMintAmount invalid as smaller than currentMintAmount"
111         );
112         inflationaryModelPerYear[yearID].maxMintAmount = maxMintAmount;
113         inflationaryModelPerYear[yearID].remainingMintAmount =
114             maxMintAmount -
115             inflationaryModelPerYear[yearID].currentMintAmount;
116
117         emit EvtUpdateInflationaryModelPerYear(yearID, maxMintAmount);
118     }

```

Listing 4.2: FBS\$ Token Contract

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged accounts are 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 accounts 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 en-

sure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated. The team confirms that the related privileged accounts will be assigned to multi-sig contract.



5 | Conclusion

In this security audit, we have examined the design and implementation of the FBS\$ 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 one informational suggestion and one low-severity issue. 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 CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [4] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [5] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [6] PeckShield. PeckShield Inc. <https://www.peckshield.com>.