SMART CONTRACT AUDIT REPORT

for

FatAnimal FAT

Prepared By: Yiqun Chen

PeckShield

July 22, 2021

Document Properties

Client	FatAnimal.finance
Title	Smart Contract Audit Report
Target	FAT
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	July 22, 2021	Jing Wang	Final Release
1.0-rc	July 21, 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 **FAT** 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 some issues related to ERC20-compliance, security, or performance. This document outlines our audit results.

1.1 About FAT

FAT is an ERC20-compliant token developed using the excellent smart contract bases from `OpenZeppelin` and `Compound`. The main features of `FAT` include the ull ERC20 compatibility and the voting power for the `FAT` token holders and their delegates.

The basic information of FAT is as follows:

Table 1.1: Basic Information of FAT

Item	Description
Issuer	FatAnimal.finance
Website	https://fatanimal.finance
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	July 22, 2021

In the following, we show the list of reviewed contracts used in this audit:

- https://bscscan.com/address/0x73280e2951785f17acc6cb2a1d0c4d65031d54b3#code

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

- 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 FAT 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	1	■
Medium	1	■
Low	0	
Informational	1	■
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 high-severity vulnerability, 1 medium-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key FAT Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Voting Amplification With Sybil Attacks	Business Logics	Confirmed
PVE-002	Medium	Trust Issue Of Admin Roles	Security Features	Fixed
PVE-003	Informational	Consistency Between Function Definitions And Return Statements	Coding Practices	Confirmed

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 FAT. 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-adopted ERC20

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()	✓

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 Voting Amplification With Sybil Attacks

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `FatAnimalToken`
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [3]

Description

The `FAT` tokens can be used for governance in allowing for users to cast and record the votes. Moreover, the `FAT` contract allows for dynamic delegation of a voter to another, though the delegation is not transitive. When a submitted proposal is being tallied, the number of votes are counted via `getPriorVotes()`.

Our analysis shows that the current governance functionality is vulnerable to a new type of so-called `Sybil` attacks. For elaboration, let's assume at the very beginning there is a malicious actor named `Malice`, who owns 100 `FAT` tokens. `Malice` has an accomplice named `Trudy` who currently has 0 balance of `FATs`. This `Sybil` attack can be launched as follows:

```
1029    function _delegate(address delegator, address delegatee)
1030        internal
1031    {
1032        address currentDelegate = _delegates[delegator];
1033        uint256 delegatorBalance = balanceOf(delegator); // balance of underlying FATs (
                not scaled);
1034        _delegates[delegator] = delegatee;
1035
1036        emit DelegateChanged(delegator, currentDelegate, delegatee);
1037
1038        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
1039    }
1040
1041    function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
1042        if (srcRep != dstRep && amount > 0) {
```

```
1043            if (srcRep != address(0)) {
1044                // decrease old representative
1045                uint32 srcRepNum = numCheckpoints[srcRep];
1046                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].
                        votes : 0;
1047                uint256 srcRepNew = srcRepOld.sub(amount);
1048                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
1049            }
1050
1051            if (dstRep != address(0)) {
1052                // increase new representative
1053                uint32 dstRepNum = numCheckpoints[dstRep];
1054                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].
                        votes : 0;
1055                uint256 dstRepNew = dstRepOld.add(amount);
1056                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
1057            }
1058        }
1059    }
```

Listing 4.1: FatAnimalToken.sol

1. Malice initially delegates the voting to Trudy. Right after the initial delegation, Trudy can have 100 votes if he chooses to cast the vote.

2. Malice transfers the full 100 balance to M_1 who also delegates the voting to Trudy. Right after this delegation, Trudy can have 200 votes if he chooses to cast the vote. The reason is that the FAT contract's transfer() does NOT _moveDelegates() together. In other words, even now Malice has 0 balance, the initial delegation (of Malice) to Trudy will not be affected, therefore Trudy still retains the voting power of 100 FATs. When M_1 delegates to Trudy, since M_1 now has 100 FATs, Trudy will get additional 100 votes, totaling 200 votes.

3. We can repeat by transferring M_i's 100 FAT balance to M_{i+1} who also delegates the votes to Trudy. Every iteration will essentially add 100 voting power to Trudy. In other words, we can effectively amplify the voting powers of Trudy arbitrarily with new accounts created and iterated!

Recommendation To mitigate, it is necessary to accompany every single transfer() and transferFrom() with the _moveDelegates() so that the voting power of the sender's delegate will be moved to the destination's delegate. By doing so, we can effectively mitigate the above Sybil attacks. Since the contract is already deployed, it is safe and acceptable to deploy another contract for governance, and use the current one for other ERC-20 functions only. A cleaner solution would be to migrate the current contract to a new one with the suggested fix, but the migration effort may be costly.

Status The issue has been confirmed by the team. The team clarifies that the voting feature is currently not used.

4.2 Trust Issue Of Admin Roles

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `FatAnimalToken`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `FatAnimalToken` 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 tokens).

To elaborate, we show below the `mint()` function in the `FatAnimalToken` contract. This function allows the `owner` to mint any amount of `FATs` to any user account.

```
855     function mint(address _to, uint256 _amount) public onlyOwner {
856       _mint(_to, _amount);
857       _moveDelegates(address(0), _delegates[_to], _amount);
858     }
```

Listing 4.2: `FatAnimalToken`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the admin roles may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among contract users.

Recommendation Make the list of extra privileges granted to `owner` explicit to FAT users.

Status This issue has been resolved by transferring the ownership from the `owner` account to the `MasterFatAnimal` contract (deployed at 0x6d4b066714cc25c9690a1c3824f7bf937df7e928), which cannot mint more `FATs`.

4.3 Consistency Between Function Definitions And Return Statements

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `FatAnimalToken`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the `FatAnimalToken` contract, the `delegate()/delegateBySig()` functions are used to delegate votes from `msg.sender` or an intended signatory to the given `delegatee`. As shown in lines 914 in the following code snippet, the `delegate()` function returns by calling the internal routine `_delegate()`, and the `return` keyword is not needed as the function is declared without any return value.

```
910  /**
911  * @notice Delegate votes from `msg.sender` to `delegatee`
912  * @param delegatee The address to delegate votes to
913  */
914  function delegate(address delegatee) external {
915      return _delegate(msg.sender, delegatee);
916  }
917
918  /**
919   * @notice Delegates votes from signatory to `delegatee`
920   * @param delegatee The address to delegate votes to
921   * @param nonce The contract state required to match the signature
922   * @param expiry The time at which to expire the signature
923   * @param v The recovery byte of the signature
924   * @param r Half of the ECDSA signature pair
925   * @param s Half of the ECDSA signature pair
926   */
927  function delegateBySig(
928      address delegatee,
929      uint nonce,
930      uint expiry,
931      uint8 v,
932      bytes32 r,
933      bytes32 s
934  )
935      external
936  {
937      bytes32 domainSeparator = keccak256(
938          abi.encode(
939              DOMAIN_TYPEHASH,
940              keccak256(bytes(name())),
941              getChainId(),
```

```
942                 address(this)
943             )
944         );
945
946         bytes32 structHash = keccak256(
947             abi.encode(
948                 DELEGATION_TYPEHASH,
949                 delegatee,
950                 nonce,
951                 expiry
952             )
953         );
954
955         bytes32 digest = keccak256(
956             abi.encodePacked(
957                 "\x19\x01",
958                 domainSeparator,
959                 structHash
960             )
961         );
962
963         address signatory = ecrecover(digest, v, r, s);
964         require(signatory != address(0), "FAT::delegateBySig: invalid signature");
965         require(nonce == nonces[signatory]++, "FAT::delegateBySig: invalid nonce");
966         require(now <= expiry, "FAT::delegateBySig: signature expired");
967         return _delegate(signatory, delegatee);
968     }
```

Listing 4.3: FatAnimalToken.sol

The same issue is also applicable for the `delegateBySig()` function.

Recommendation Remove the `return` keyword in the above two functions. An example revision is shown as follows.

```
910     /**
911     * @notice Delegate votes from `msg.sender` to `delegatee`
912     * @param delegatee The address to delegate votes to
913     */
914     function delegate(address delegatee) external {
915         _delegate(msg.sender, delegatee);
916     }
917
918     /**
919      * @notice Delegates votes from signatory to `delegatee`
920      * @param delegatee The address to delegate votes to
921      * @param nonce The contract state required to match the signature
922      * @param expiry The time at which to expire the signature
923      * @param v The recovery byte of the signature
924      * @param r Half of the ECDSA signature pair
925      * @param s Half of the ECDSA signature pair
926      */
927     function delegateBySig(
```

```
928         address delegatee,
929         uint nonce,
930         uint expiry,
931         uint8 v,
932         bytes32 r,
933         bytes32 s
934     )
935         external
936     {
937         bytes32 domainSeparator = keccak256(
938             abi.encode(
939                 DOMAIN_TYPEHASH,
940                 keccak256(bytes(name())),
941                 getChainId(),
942                 address(this)
943             )
944         );
945 
946         bytes32 structHash = keccak256(
947             abi.encode(
948                 DELEGATION_TYPEHASH,
949                 delegatee,
950                 nonce,
951                 expiry
952             )
953         );
954 
955         bytes32 digest = keccak256(
956             abi.encodePacked(
957                 "\x19\x01",
958                 domainSeparator,
959                 structHash
960             )
961         );
962 
963         address signatory = ecrecover(digest, v, r, s);
964         require(signatory != address(0), "FAT::delegateBySig: invalid signature");
965         require(nonce == nonces[signatory]++, "FAT::delegateBySig: invalid nonce");
966         require(now <= expiry, "FAT::delegateBySig: signature expired");
967         _delegate(signatory, delegatee);
968     }
```

Listing 4.4: FatAnimalToken.sol (revised)

Status This issue has been confirmed. Considering the fact that this contract has been deployed and this finding does not affect any normal functionalities, the team prefers not modifying the code and leaves it as is.

5 | Conclusion

In this security audit, we have examined the design and implementation of the FAT 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 two 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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