BEOSIN
Blockchain Security

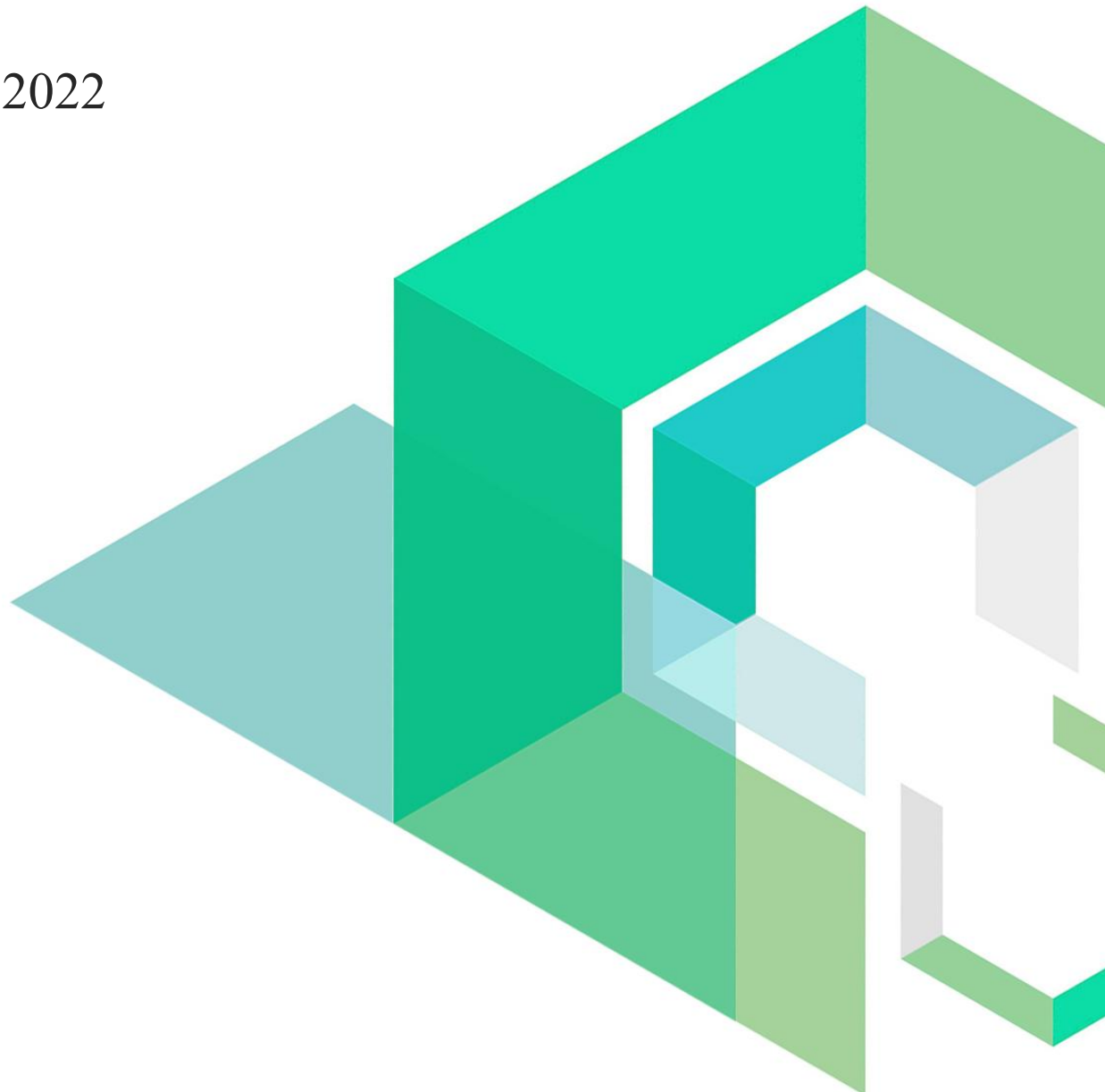
Avalanche

Smart Contract Security Audit

V1.0

No. 202207121350

Jul 12nd, 2022



Contents

Summary of audit results **1**

1 Overview **3**

1.1 Project Overview 3

1.2 Audit Overview 3

2 Findings **4**

[Avalanche-1] Owner and Operator have a high permission 5

[Avalanche-2] Unusual amount of burned tokens 7

[Avalanche-3] Wrong balance used 8

[Avalanche-4] Denial of Service Attack Risk 9

[Avalanche-5] Add the same data to the array repeatedly 11

[Avalanche-6] Centralization Risk 12

[Avalanche-7] The event trigger does not match the actual number 14

3 Appendix **15**

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts 15

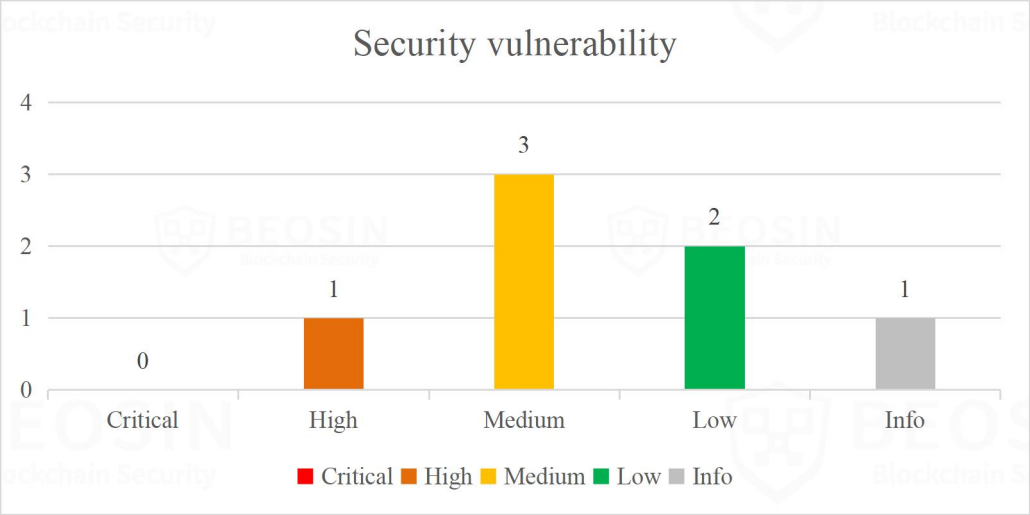
3.2 Audit Categories 17

3.3 Disclaimer 19

3.4 About BEOSIN 20

Summary of audit results

After auditing, 1 High-risk, 3 Medium-risk, 2 Low-risk and 1 Info items were identified in the Avalanche project. Specific audit details will be presented in the **Findings section**. Users should pay attention to the following aspects when interacting with this project:



*Notes:

- **Risk Description:**

1. Centralization risk

The owner can set the operator address, and both the owner and operator addresses can modify key parameters in the contract. For some parameters, only the owner has the permission to modify. There may be some centralization risk.

2. Risk of insufficient gas

Multiple for loops are used in many places in the contract. If there are too many loops, the related function calls may fail.

3. User withdrawal risk

When users withdraw staked tokens from the AvalanchePool contract, they first need to apply, and then the operator address will issue them. If the operator address does not call the *serveClaims* function or does not pass in enough AVAX tokens, the user may not be able to withdraw as expected.

- **Project Description:**

1. Basic Token Information

Token name	Ankr Avalanche Reward Bearing Certificate
Token symbol	aAVAXc
Decimals	18
Pre-mint	0
Total supply	Initial supply is 0 (Mintable, burnable)
Token type	ERC-20

Table 1 aAVAXc token info

Token name	Ankr Avalanche Reward Earning Bond
Token symbol	aAVAXb
Decimals	18
Pre-mint	0
Total supply	Initial supply is 0 (Mintable, burnable)
Token type	ERC-20

Table 2 aAVAXb token info

2. Business overview

The Avalanche project contains two token contracts and one business contracts. In the token contract, the number of shares is recorded inside the contract, and what the user queries is the number of bonds. Shares and bonds are converted according to a certain ratio (the ratio can be arbitrarily modified by the owner or operator address). Users can stake AVAX tokens in the AvalanchePool contract to obtain aAVAXb tokens, and the aAVAXb tokens and aAVAXc tokens are interchangeable on a 1:1 ratio. When the users withdraw the AVAX tokens staked in the AvalanchePool contract, they need to first apply for the withdrawal, and then the operator address calls the *serveClaims* function to send the AVAX tokens to the user.

1 Overview

1.1 Project Overview

Project Name	Avalanche	
Platform	Avalanche C-Chain	
File Hash (SHA256)	FutureBondAVAX.sol	d079c8a0cb045aae71d5b9838dbc2b60762a0d77676d296c5015d1ecd6412a8d (Initial) 01e87a28e0e0cc1d8a24077e9de210a9666a6f6150ab6b5bd0be99af5268351c (Final)
	ERC20Bond.sol	9a1cb553d096174761689ac666beaba184e1a90172d913abdfcf6fd6c8a0d12d
	AvalanchePool.sol	966992b8581529734edbb5cc69aaca7f399ce0542d070a65cefab7512f65cc10 (Initial) ca5a28b55f210ed450d5c22f92f8302f0dd17e95624345acad2038780da6a5d9 (Final)
	FutureCertAVAX.sol	ad6307d62303b12582be56bb4ac46f3a1fbc37a6d54424e09ee1fd80ac6881a1

1.2 Audit Overview

Audit work duration: June 6, 2022 – July 12, 2022

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Technology Co. Ltd.

2 Findings

Index	Risk description	Severity level	Status
Avalanche-1	Owner and Operator have a high permission	High	Fixed
Avalanche-2	Unusual amount of burned tokens	Medium	Fixed
Avalanche-3	Wrong balance used	Medium	Fixed
Avalanche-4	Denial of Service Attack Risk	Medium	Fixed
Avalanche-5	Add the same data to the array repeatedly	Low	Fixed
Avalanche-6	Centralization Risk	Low	Acknowledged
Avalanche-7	The event trigger does not match the actual number	Info	Fixed

Risk Details Description:

1. Avalanche-6 is not fixed and may cause a certain centralization risk.

[Avalanche-1] Owner and Operator have a high permission

Severity Level	**High**
Type	Business Security
Lines	FutureBondAVAX.sol#L117-125, L132-136, L147-159, L274-287
Description	The Owner address in the FutureBondAVAX contract can call *mint* and *mintBonds* functions to mint tokens to any address, and *burn*, *lockForDelayedBurn* and *commitDelayedBurn* functions to destroy any address tokens. There is a problem that the owner permission is too large.

```
117     function mintBonds(address account, uint256 amount) public override onlyBondMinter {
118         uint256 shares = _bondsToShares(amount);
119         _mint(account, shares);
120     }
121
122     function mint(address account, uint256 shares) public onlyMinter {
123         _lockedShares = _lockedShares.sub(int256(shares));
124         _mint(account, shares);
125     }
```

Figure 1 Source code of related functions

```
132     function burn(address account, uint256 amount) public override onlyMinter {
133         uint256 shares = _bondsToShares(amount);
134         _lockedShares = _lockedShares.add(int256(shares));
135         _burn(account, shares);
136     }
```

Figure 2 Source code of burn function

```
147     function lockForDelayedBurn(address account, uint256 amount) public override onlyBondMinter {
148         _pendingBurn[account] = _pendingBurn[account].add(amount);
149         _pendingBurnsTotal = _pendingBurnsTotal.add(amount);
150     }
151
152     function commitDelayedBurn(address account, uint256 amount) public override onlyBondMinter {
153         uint256 burnableAmount = _pendingBurn[account];
154         require(burnableAmount >= amount, "Too big amount to burn");
155         uint256 sharesToBurn = _fAvaxToSharesConfirmedRatio(amount);
156         _pendingBurn[account] = burnableAmount.sub(amount);
157         _pendingBurnsTotal = _pendingBurnsTotal.sub(amount);
158         _burn(account, sharesToBurn);
159     }
```

Figure 3 Source code of related functions

```
274  modifier onlyOperator() {
275      require(msg.sender == owner() || msg.sender == _operator, "Operator: not allowed");
276      _;
277  }
278
279  modifier onlyMinter() {
280      require(msg.sender == owner() || msg.sender == _crossChainBridge, "Minter: not allowed");
281      _;
282  }
283
284  modifier onlyBondMinter() {
285      require(msg.sender == owner() || msg.sender == _avalanchePool, "Minter: not allowed");
286      _;
287  }
```

Figure 4 Source code of related modifiers (Unfixed)

Recommendations	It is recommended to remove the owner's minting and burning permissions.
Status	Fixed.

```
265  modifier onlyOperator() {
266      require(msg.sender == _operator, "Operator: not allowed");
267      _;
268  }
269
270  modifier onlyMinter() {
271      require(msg.sender == _crossChainBridge, "Minter: not allowed");
272      _;
273  }
274
275  modifier onlyBondMinter() {
276      require(msg.sender == _avalanchePool, "Minter: not allowed");
277      _;
278  }
```

Figure 5 Source code of related modifiers (Fixed)

[Avalanche-2] Unusual amount of burned tokens

Severity Level	Medium
Type	Business Security
Lines	FutureBondAVAX.sol#L152-159, L270-272
Description	In the FutureBondAVAX contract, when the *balanceOf* function queries the number of tokens held by the specified address, ratio is used when converting shares to bonds, but in the *commitDelayedBurn* function, lastConfirmedRatio is used when burning shares. If ratio and lastConfirmedRatio are not equal, the number of bonds queried by the user before and after the *commitDelayedBurn* function is called may change.

```
152    function commitDelayedBurn(address account, uint256 amount) public override onlyBondMinter {
153        uint256 burnableAmount = _pendingBurn[account];
154        require(burnableAmount >= amount, "Too big amount to burn");
155        uint256 sharesToBurn = _fAvaxToSharesConfirmedRatio(amount);
156        _pendingBurn[account] = burnableAmount.sub(amount);
157        _pendingBurnsTotal = _pendingBurnsTotal.sub(amount);
158        _burn(account, sharesToBurn);
159    }
```

Figure 6 Source code of *commitDelayedBurn* function (Unfixed)

```
270    function _fAvaxToSharesConfirmedRatio(uint256 amount) internal view returns (uint256) {
271        return safeMultiplyAndDivide(amount, _lastConfirmedRatio, 1e18);
272    }
```

Figure 7 Source code of *_fAvaxToSharesConfirmedRatio* function

Recommendations	It is recommended to also use ratio to calculate the quantity when burning.
Status	Fixed.

```
146    function commitDelayedBurn(address account, uint256 amount) public override onlyBondMinter {
147        uint256 burnableAmount = _pendingBurn[account];
148        require(burnableAmount >= amount, "Too big amount to burn");
149        uint256 sharesToBurn = _bondsToShares(amount);
150        _pendingBurn[account] = burnableAmount.sub(amount);
151        _pendingBurnsTotal = _pendingBurnsTotal.sub(amount);
152        _burn(account, sharesToBurn);
153    }
```

Figure 8 Source code of *commitDelayedBurn* function (Fixed)

[Avalanche-3] Wrong balance used

Severity Level	Medium
Type	Business Security
Lines	FutureBondAVAX.sol#L209-223
Description	In the FutureBondAVAX contract, the *_unlockShares* function uses "super.balanceOf(account)" to calculate the balance, but the balance in the pendingBurn state is not excluded.

```
209     function _unlockShares(address account, uint256 shares, bool takeFee) internal {
210         require(super.balanceOf(account) >= shares, "Insufficient aAVAXb balance");
211
212         uint256 fee = 0;
213         if (takeFee) {
214             fee = getSwapFeeInShares(shares);
215         }
216
217         transferShares(account, address(this), shares - fee);
218         if (fee != 0) {
219             transferShares(account, _swapFeeOperator, fee);
220         }
221
222         ICertAVAX(_certToken).bondTransferTo(account, shares - fee);
223     }
```

Figure 9 Source code of *_unlockShares* function (Unfixed)

Recommendations	It is recommended to exclude the balance in the pendingBurn state.
Status	Fixed.

```
204     function _unlockShares(address account, uint256 shares, bool takeFee) internal {
205         require(balanceOf(account) >= _sharesToBonds(shares), "Insufficient aAVAXb balance");
206
207         uint256 fee = 0;
208         if (takeFee) {
209             fee = getSwapFeeInShares(shares);
210         }
211
212         transferShares(account, address(this), shares - fee);
213         if (fee != 0) {
214             transferShares(account, _swapFeeOperator, fee);
215         }
216
217         ICertAVAX(_certToken).bondTransferTo(account, shares - fee);
218     }
```

Figure 10 Source code of *_unlockShares* function (Fixed)

[Avalanche-4] Denial of Service Attack Risk

Severity Level	Medium
Type	Business Security
Lines	AvalanchePool.sol#L230-280
Description	In the AvalanchePool contract, when the *serveClaims* function calls wallet.transfer, if the wallet is a contract and refuses to accept the platform token, it may cause a DOS attack.

```
230     function serveClaims(address payable residueAddress, uint256 minThreshold) public onlyOperator payable {
231         address[] memory claimers = new address[](_pendingClaimers.length.sub(_pendingAvaxClaimGap));
232         uint256[] memory amounts = new uint256[](_pendingClaimers.length.sub(_pendingAvaxClaimGap));
233         uint256 availableAmount = msg.value;
234         uint256 j = 0;
235         uint256 gaps = 0;
236         uint256 i = 0;
237         for (i = _pendingAvaxClaimGap; i < _pendingClaimers.length; i++) {
238             /* if the number of tokens left is less than threshold do not try to serve the claims */
239             if (availableAmount < minThreshold) {
240                 break;
241             }
242             address claimer = _pendingClaimers[i];
243             uint256 amount = _pendingUserClaims[claimer];
244             /* we might have gaps lets just skip them (we shrink them on full claim) */
245             if (claimer == address(0) || amount == 0) {
246                 gaps++;
247                 continue;
248             }
249             if (availableAmount < amount) {
250                 break;
251             }
252             claimers[j] = claimer;
253             amounts[j] = amount;
254             address payable wallet = payable(address(claimer));
255             wallet.transfer(amount);
256             availableAmount = availableAmount.sub(amount);
257             j++;
258             IBondAVAX(_fAvaxContract).commitDelayedBurn(claimer, amount);
259             _pendingUserClaims[claimer] = 0;
260             delete _pendingClaimers[i];
261             /* when we delete items from array we generate new gap, lets remember how many gaps we did to skip them in next claim */
262             gaps++;
263         }
```

Figure 11 Source code of *serveClaims* function (Unfixed)

Recommendations	It is recommended that the address to call *_claim* function cannot be the contract address.
Status	Fixed.

```
244     function serveClaims(address payable residueAddress, uint256 minThreshold) public onlyOperator payable {
245         address[] memory claimers = new address[](_pendingClaimers.length.sub(_pendingAvaxClaimGap));
246         uint256[] memory amounts = new uint256[](_pendingClaimers.length.sub(_pendingAvaxClaimGap));
247         uint256 availableAmount = msg.value;
248         uint256 j = 0;
249         uint256 gaps = 0;
250         uint256 i = 0;
251         for (i = _pendingAvaxClaimGap; i < _pendingClaimers.length; i++) {
252             /* if the number of tokens left is less than threshold do not try to serve the claims */
253             if (availableAmount < minThreshold) {
254                 break;
255             }
256             address claimer = _pendingClaimers[i];
257             uint256 amount = _pendingUserClaims[claimer];
258             /* we might have gaps lets just skip them (we shrink them on full claim) */
259             if (claimer == address(0) || amount == 0) {
260                 gaps++;
261                 continue;
262             }
263             if (availableAmount < amount) {
264                 break;
265             }
266             address payable wallet = payable(address(claimer));
267             bool result = wallet.send(amount);
268             if (!result) {
269                 _stashedForManualClaim = _stashedForManualClaim.add(amount);
270                 _manualClaimers[i] = claimer;
271                 _manualStashed[i] = amount;
272                 emit ManualClaimExpected(claimer, amount, i);
273             } else {
274                 claimers[j] = claimer;
275                 amounts[j] = amount;
276                 j++;
277                 IBondAVAX(_fAvaxContract).commitDelayedBurn(claimer, amount);
278             }
279             availableAmount = availableAmount.sub(amount);
280             _pendingUserClaims[claimer] = 0;
281             delete _pendingClaimers[i];
282             /* when we delete items from array we generate new gap, lets remember how many gaps we did to skip them in next claim */
283             gaps++;
284         }
285         _pendingAvaxClaimGap = _pendingAvaxClaimGap.add(gaps);
286         uint256 missing = 0;
287         for (i = _pendingAvaxClaimGap; i < _pendingClaimers.length; i++) {
288             missing = missing.add(_pendingUserClaims[_pendingClaimers[i]]);
289         }
```

Figure 12 Source code of *serveClaims* function (Fixed)

[Avalanche-5] Add the same data to the array repeatedly

Severity Level	**Low**
Type	Business Security
Lines	AvalanchePool.sol#L219-228
Description	In the AvalanchePool contract, if the amount input by the *_claim* function is 0, function can repeatedly push _pendingCaimers.

```
219     function _claim(uint256 amount, bool isRebasing) internal {
220         require(IERC20Upgradeable(_fAvaxContract).balanceOf(msg.sender) >= amount, "Cannot claim more than have on address");
221         if (_pendingUserClaims[msg.sender] == 0) {
222             _pendingClaimers.push(msg.sender);
223         }
224         _pendingUserClaims[msg.sender] = _pendingUserClaims[msg.sender].add(amount);
225         IBondAVAX(_fAvaxContract).lockForDelayedBurn(msg.sender, amount);
226         emit AvaxClaimPending(msg.sender, amount);
227         emit AvaxClaimPendingV2(msg.sender, amount, isRebasing);
228     }
```

Figure 13 Source code of *_claim* function (Unfixed)

Recommendations	It is recommended to judge whether the number is greater than 0.
Status	Fixed.

```
224     function _claim(uint256 amount, bool isRebasing) internal {
225         require(amount > 0, "AvalanchePool: cannot claim zero");
226         require(IERC20Upgradeable(_fAvaxContract).balanceOf(msg.sender) >= amount, "Cannot claim more than have on address");
227         if (_pendingUserClaims[msg.sender] == 0) {
228             _pendingClaimers.push(msg.sender);
229         }
230         _pendingUserClaims[msg.sender] = _pendingUserClaims[msg.sender].add(amount);
231         IBondAVAX(_fAvaxContract).lockForDelayedBurn(msg.sender, amount);
232         emit AvaxClaimPending(msg.sender, amount);
233         emit AvaxClaimPendingV2(msg.sender, amount, isRebasing);
234     }
```

Figure 14 Source code of *_claim* function (Fixed)

[Avalanche-6] Centralization Risk

Severity Level	**Low**
Type	Business Security
Lines	FutureBondAVAX.sol#L90-102, L55-84
Description	The owner and operator in the FutureBondAVAX contract can call *setNameAndSymbol*, *updateRatio*, *updateLastConfirmedRatio*, *updateBothRatios*, *updateBothRatiosAndFee* and other functions to modify some related parameters of the contract. The Owner address can also call functions such as *changeOperator*, *changeAvalanchePool*, *changeCrossChainBridge*, *changeCertToken*, *changeSwapFeeOperator*, *updateSwapFeeRatio*, *repairCollectableFee*, and *repairRatios* to modify some contract parameters. There may be some centralization risk.

```
90      function repairCollectableFee(uint256 newFee) public onlyOwner {
91          _collectableFee = newFee;
92      }
93
94      function repairRatios(uint256 newRatio, uint256 newConfirmedRatio) public onlyOwner {
95          _ratio = newRatio;
96          _lastConfirmedRatio = newConfirmedRatio;
97      }
98
99      function totalSupply() public view override returns (uint256) {
100         uint256 supply = totalSharesSupply();
101         return _sharesToBonds(supply);
102     }
```

Figure 15 Source code of related functions

```
55  function updateRatio(uint256 newRatio) public onlyOperator {
56      // 0.002 * ratio
57      uint256 threshold = _ratio.div(500);
58      require(newRatio < _ratio.add(threshold) || newRatio > _ratio.sub(threshold), "New ratio should be in limits");
59      _ratio = newRatio;
60      emit RatioUpdate(_ratio);
61  }
62
63  function lastConfirmedRatio() public view override returns (uint256) {
64      return _lastConfirmedRatio;
65  }
66
67  function updateLastConfirmedRatio(uint256 newRatio) public onlyOperator {
68      // 0.002 * ratio
69      uint256 threshold = _lastConfirmedRatio.div(500);
70      require(newRatio < _lastConfirmedRatio.add(threshold) || newRatio > _lastConfirmedRatio.sub(threshold), "New ratio should be in limits");
71      _lastConfirmedRatio = newRatio;
72      emit LastConfirmedRatioUpdate(_lastConfirmedRatio);
73  }
74
75  function updateBothRatios(uint256 newRatio, uint256 newConfirmedRatio) public onlyOperator {
76      updateRatio(newRatio);
77      updateLastConfirmedRatio(newConfirmedRatio);
78  }
79
80  function updateBothRatiosAndFee(uint256 newRatio, uint256 newConfirmedRatio, uint256 newFee) public onlyOperator {
81      updateRatio(newRatio);
82      updateLastConfirmedRatio(newConfirmedRatio0);
83      _collectableFee = newFee;
84  }
```

Figure 16 Source code of related functions

Recommendations	It is recommended to use multi-signature wallet, DAO, TimeLock contract,

	etc. as the contract owner.
Status	Acknowledged.

[Avalanche-7] The event trigger does not match the actual number

Severity Level	Info
Type	Coding Conventions
Lines	FutureBondAVAX.sol#L161-165, L231-251
Description	The number of tokens in events such as token transfer and authorization in the FutureBondAVAX contract is share, but what the user queried is bonds.

```
161  function transfer(address recipient, uint256 amount) public override returns (bool) {
162      uint256 shares = _bondsToSharesCeil(amount);
163      super.transfer(recipient, shares);
164      return true;
165  }
```

Figure 17 Source code of transfer function

```
231  function _transfer(
232      address from,
233      address to,
234      uint256 amount
235  ) internal virtual {
236      require(from != address(0), "ERC20: transfer from the zero address");
237      require(to != address(0), "ERC20: transfer to the zero address");
238
239      _beforeTokenTransfer(from, to, amount);
240
241      uint256 fromBalance = _balances[from];
242      require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");
243      unchecked {
244          _balances[from] = fromBalance - amount;
245      }
246      _balances[to] += amount;
247
248      emit Transfer(from, to, amount);
249
250      _afterTokenTransfer(from, to, amount);
251  }
```

Figure 18 Source code of *transfer* function (Unfixed)

Recommendations	It is recommended to modify the number of tokens in the relevant event to bonds.
Status	Fixed.

```
88  function _transfer(address sender, address recipient, uint256 amount) internal virtual {
89      require(sender != address(0), "ERC20: transfer from the zero address");
90      require(recipient != address(0), "ERC20: transfer to the zero address");
91
92      _beforeTokenTransfer(sender, recipient, amount);
93
94      _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
95      _balances[recipient] = _balances[recipient].add(amount);
96      emit Transfer(sender, recipient, _sharesToBonds(amount));
97  }
```

Figure 19 Source code of *transfer* function (Fixed)

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Likelihood \ Impact	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	High	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

3.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in Blockchain.

3.4 About BEOSIN

Affiliated to BEOSIN Technology Pte. Ltd., BEOSIN is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions.BEOSIN has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, BEOSIN has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website

https://www.beosin.com

Telegram

https://t.me/+dD8Bnqd133RmNWNl

Twitter

https://twitter.com/Beosin_com

Email

Contact@beosin.com