BLOCKSEC

Security Audit
Report for KBAR

Date: April 8, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	DuboisGold
Target	KBAR

Version History

Version	Date	Description
1.0	April 8, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-04-16 02:52:44+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of KBAR of DuboisGold.

KBAR is an upgradeable ERC721 contract representing ownership of physical gold bars held in custody. The system integrates role-based administration with transfer-time compliance controls and fee mechanisms. Administrative roles manage minting, redemption, freezing, fee configuration, storage-fee enforcement, pausing, and upgrades.

Note this audit only focuses on the smart contracts in the following directories/files:

- kbar-v2/src/KBAR.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
KBAR	`Version 1`	`ada693c9114f9e73cf8bdaae8bc559d3763a4947`
	`Version 2`	`6eda334c88d573b7ecffe038a200224a95731f58`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not

[1] https://github.com/DuboisGold/contracts-v2

guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

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

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **two** potential security issues. Besides, we have **two** recommendations and **one** note.

- Low Risk: 2
- Recommendation: 2
- Note: 1

ID	Severity	Description	Category	Status
1	Low	Lack of operator freeze check in the function `_update()`	Security Issue	Fixed
2	Low	Unfair storage fee accrual logic	Security Issue	Confirmed
3	-	Inconsistent transfer eligibility validation	Recommendation	Partially Fixed
4	-	Add recipient freeze check to the function `safeMint()`	Recommendation	Fixed
5	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of operator freeze check in the function `_update()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_update()` enforces freeze checks on `sender`, `recipient`, and `token`, but does not validate the operator represented by `auth`. A frozen address can still execute transfers as an approved operator on behalf of other users, circumventing the intended restriction imposed by the function `freezeAddress()`.

The freeze mechanism therefore does not fully restrict participation of frozen addresses in transfer execution.

```
968   function _update(
969       address to,
970       uint256 tokenId,
971       address auth
972   ) internal override(ERC721Upgradeable, ERC721EnumerableUpgradeable) returns (address) {
973       address from = _ownerOf(tokenId);
974
975       if (from != address(0) && to != address(0)) {
976           if (paused()) revert EnforcedPause();
977           if (_frozenAddresses[from]) revert AddressIsFrozen(from);
978           if (_frozenAddresses[to]) revert AddressIsFrozen(to);
979           if (_frozenTokens[tokenId]) revert TokenIsFrozen(tokenId);
980
981           // Storage fee enforcement (only when activated by admin).
```

```
982             // Custody / operational addresses marked storageEnforcementExempt bypass
983             // the overdue check - their transfers proceed without modifying paidUntil.
984             if (enforcementStart != 0 && block.timestamp >= enforcementStart) {
985                 if (!storageEnforcementExempt[from] && !storageEnforcementExempt[to]) {
986                     if (_storageFeeStates[tokenId].paidUntil < block.timestamp) {
987                         revert TransferBlockedStorageFeesDue(tokenId);
988                     }
989                 }
990             }
991         }
992 
993         return super._update(to, tokenId, auth);
994     }
```

Listing 2.1: packages/kbar-v2/src/KBAR.sol

```
609     /// @notice Freeze an address - prevents it from sending or receiving KBARs
610     /// @param account Address to freeze
611     function freezeAddress(
612         address account
613     ) external onlyRole(FREEZER_ROLE) {
614         if (account == address(0)) revert InvalidParameter("account");
615         if (_frozenAddresses[account]) revert AlreadyFrozen();
616         _frozenAddresses[account] = true;
617         emit AddressFrozen(account);
618     }
```

Listing 2.2: packages/kbar-v2/src/KBAR.sol

Impact Frozen addresses can continue to influence token movements through operator permissions.

Suggestion Extend the freeze validation to include the operator address used in the transfer path.

2.1.2 Unfair storage fee accrual logic

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `KBAR` contract, the function `payStorageFee()` permits storage-fee prepayment before `enforcementStart` becomes active. Once a token has a non-zero `paidUntil`, the function `_storageFeeStart()` always uses that historical `paidUntil` as the accrual starting point. It does not reset the accrual baseline to `enforcementStart` after storage-fee enforcement is later enabled.

As a result, when a token was prepaid before enforcement but its recorded `paidUntil` still remains earlier than the eventual `enforcementStart`, the token holder must continue paying from that earlier historical timestamp to restore transferability after enforcement begins. In contrast, if no prepayment had been made and `paidUntil` had remained zero, the first payment

after enforcement would start from `enforcementStart`. This creates an unfair outcome where prior prepayment can lead to a less favorable payment baseline than no prepayment at all.

```
843         if (enforcementStart != 0 && nowTs >= enforcementStart && newPaidUntil < nowTs) {
```

Listing 2.3: packages/kbar-v2/src/KBAR.sol

```
815     function payStorageFee(
816         uint256 tokenId,
817         uint256 daysToPay,
818         IERC20 paymentToken
819     ) external nonReentrant {
820         if (!_storageFeeConfigured()) revert StorageFeeNotConfigured();
821         if (address(paymentToken) != address(usdc) && address(paymentToken) != address(usdt)) {
822             revert InvalidPaymentToken();
823         }
824 
825         uint256 feeAmount = calculateStorageFee(tokenId, daysToPay);
826         if (feeAmount == 0) revert FeeBelowMinimum();
827 
828         // forge-lint: disable-next-line(unsafe-typecast)
829         uint48 nowTs = uint48(block.timestamp);
830 
831         uint48 start = _storageFeeStart(tokenId);
832         // forge-lint: disable-next-line(unsafe-typecast)
833         uint48 newPaidUntil = start + uint48(daysToPay * 1 days);
834 
835         if (newPaidUntil > nowTs + MAX_PAIDUNTIL_AHEAD) {
836             revert InvalidParameter("paidUntil too far");
837         }
838 
839         // When enforcement is active, reject payments that don't actually unblock the token.
840         // Prevents users from paying real money while their token remains transfer-blocked.
841         // Inline enforcementStart check instead of isStorageFeeEnforcementActive()
842         // so the entire condition uses the same nowTs snapshot.
843         if (enforcementStart != 0 && nowTs >= enforcementStart && newPaidUntil < nowTs) {
844             revert PaymentStillOverdue(newPaidUntil, nowTs);
845         }
846 
847         // State update before external call (Checks-Effects-Interactions)
848         _storageFeeStates[tokenId].paidUntil = newPaidUntil;
849 
850         paymentToken.safeTransferFrom(msg.sender, treasury, feeAmount);
851 
852         emit StorageFeePaid(tokenId, msg.sender, address(paymentToken), feeAmount, newPaidUntil);
853     }
```

Listing 2.4: packages/kbar-v2/src/KBAR.sol

```
1119    function _storageFeeStart(
1120        uint256 tokenId
1121    ) private view returns (uint48 start) {
1122        uint48 currentPaidUntil = _storageFeeStates[tokenId].paidUntil;
1123        if (currentPaidUntil != 0) {
```

```
1124            return currentPaidUntil;
1125        }
1126        if (enforcementStart != 0) {
1127            return enforcementStart;
1128        }
1129        // forge-lint: disable-next-line(unsafe-typecast)
1130        return uint48(block.timestamp);
1131    }
```

Listing 2.5: packages/kbar-v2/src/KBAR.sol

Impact Users who prepaid before enforcement may need to pay more to recover transferability than users who never prepaid, resulting in unfair storage-fee treatment.

Suggestion Revise the logic accordingly.

Feedback from the project Storage fees are intended to reflect real-world custody costs, which accrue continuously regardless of whether the token is transferable. Accordingly, when applicable, fee accrual continues from the historical `paidUntil` timestamp. `enforcementStart` only determines when transfer restrictions become effective and is not intended to reset the baseline for storage fee accrual.

2.2 Recommendation

2.2.1 Inconsistent transfer eligibility validation

Status Partially Fixed in `Version 2`

Introduced by `Version 1`

Description The helper functions `canTransfer()` and `canTransferTo()` do not fully mirror the conditions enforced during actual transfers. In addition to the paused state, the execution path also depends on transfer fee configuration, while the view functions do not validate whether transfer fees are properly configured.

As a result, these helper functions may return a transferable state even when the actual transfer would revert due to missing fee configuration or other enforced conditions.

```
320    /**
321     * @notice Check if a token can be transferred (source-side only)
322     * @dev Does NOT check the destination address - use canTransferTo() for that.
323     *      Accounts marked storageEnforcementExempt bypass the overdue check (sender side).
324     * @param tokenId Token to check
325     * @return True if the token is transferable from its current owner
326     */
327    function canTransfer(
328        uint256 tokenId
329    ) external view returns (bool) {
330        address owner = _ownerOf(tokenId);
331        if (owner == address(0)) return false;
332        if (_frozenAddresses[owner]) return false;
333        if (_frozenTokens[tokenId]) return false;
334        if (!isStorageFeeEnforcementActive()) return true;
```

```
335         if (storageEnforcementExempt[owner]) return true;
336         return _storageFeeStates[tokenId].paidUntil >= block.timestamp;
337     }
338
339     /**
340      * @notice Check if a token can be transferred to a specific recipient
341      * @dev Same as canTransfer(), plus checks that recipient is not frozen.
342      *      Accounts marked storageEnforcementExempt bypass the overdue check
343      *      (either sender or recipient being exempt is sufficient).
344      * @param tokenId Token to check
345      * @param to Intended recipient address
346      * @return True if the transfer would succeed (ignoring fee payment)
347      */
348     function canTransferTo(
349         uint256 tokenId,
350         address to
351     ) external view returns (bool) {
352         if (to == address(0)) return false;
353         address owner = _ownerOf(tokenId);
354         if (owner == address(0)) return false;
355         if (_frozenAddresses[owner]) return false;
356         if (_frozenAddresses[to]) return false;
357         if (_frozenTokens[tokenId]) return false;
358         if (!isStorageFeeEnforcementActive()) return true;
359         if (storageEnforcementExempt[owner] || storageEnforcementExempt[to]) return true;
360         return _storageFeeStates[tokenId].paidUntil >= block.timestamp;
361     }
```

Listing 2.6: packages/kbar-v2/src/KBAR.sol

Suggestion Align helper function logic with the full set of transfer restrictions to ensure consistent behavior.

Feedback from the project The helper functions are intended to serve as lightweight UI/backend checks for core logical transfer restrictions, rather than as a full runtime simulation.

Clarification from BlockSec The project team has added paused-state validation and clarified the NatSpec comments to explicitly state that these helper functions do not simulate the fee payment execution path, oracle runtime validation, operator authorization context, or ERC-721 receiver callback success.

2.2.2 Add recipient freeze check to the function `safeMint()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `safeMint()` allows minting to any non-zero address without validating whether the recipient is frozen. The freeze checks in function `_update()` only apply when both sender and recipient are non-zero, which excludes minting flows.

As a result, tokens can be assigned to frozen addresses, which contradicts the intended semantics of address freezing.

```
568   function safeMint(
569       address to,
570       string calldata uri
571   ) public onlyRole(MINTER_ROLE) returns (uint256 tokenId) {
572       if (to == address(0)) revert InvalidParameter("to");
573
574       tokenId = _nextTokenId++;
575
576       // Storage fee: only initialize paidUntil when enforcement is already active.
577       // Pre-enforcement tokens stay at 0; admin batch-records before enabling enforcement.
578       if (isStorageFeeEnforcementActive()) {
579           _storageFeeStates[tokenId].paidUntil = uint48(block.timestamp);
580       }
581
582       _safeMint(to, tokenId);
583       _setTokenURI(tokenId, uri);
584
585       emit BarMinted(tokenId, to);
586   }
```

Listing 2.7: packages/kbar-v2/src/KBAR.sol

```
968   function _update(
969       address to,
970       uint256 tokenId,
971       address auth
972   ) internal override(ERC721Upgradeable, ERC721EnumerableUpgradeable) returns (address) {
973       address from = _ownerOf(tokenId);
974
975       if (from != address(0) && to != address(0)) {
976           if (paused()) revert EnforcedPause();
977           if (_frozenAddresses[from]) revert AddressIsFrozen(from);
978           if (_frozenAddresses[to]) revert AddressIsFrozen(to);
979           if (_frozenTokens[tokenId]) revert TokenIsFrozen(tokenId);
980
981           // Storage fee enforcement (only when activated by admin).
982           // Custody / operational addresses marked storageEnforcementExempt bypass
983           // the overdue check - their transfers proceed without modifying paidUntil.
984           if (enforcementStart != 0 && block.timestamp >= enforcementStart) {
985               if (!storageEnforcementExempt[from] && !storageEnforcementExempt[to]) {
986                   if (_storageFeeStates[tokenId].paidUntil < block.timestamp) {
987                       revert TransferBlockedStorageFeesDue(tokenId);
988                   }
989               }
990           }
991       }
992
993       return super._update(to, tokenId, auth);
994   }
```

Listing 2.8: packages/kbar-v2/src/KBAR.sol

Suggestion Add a recipient freeze check in the function `safeMint()`.

2.3 Note

2.3.1 Potential centralization risks

Introduced by Version 1

Description The project relies on multiple privileged roles (e.g., DEFAULT_ADMIN_ROLE, FEE_ADMIN_ROLE, FREEZER_ROLE, REDEEMER_ROLE, BURNER_ROLE, and PAUSER_ROLE) to manage core functions including role administration, fee configuration, storage-fee enforcement, freezing, redemption, burning, pausing, and upgrades. These roles directly control token lifecycle, transferability, and economic parameters, creating a centralized control surface over the system.

In addition, the protocol depends on off-chain operational inputs to maintain consistency with the underlying gold business. Storage-fee status, enforcement timing, exemption decisions, and asset lifecycle events are determined off-chain and then reflected on-chain through privileged actions, without independent on-chain verification.

If privileged accounts are compromised or misused, or if off-chain inputs are inaccurate, incomplete, or manipulated, the protocol may be exposed to incorrect state transitions, user asset impact, and loss of operational integrity.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS