BLOCKSEC

Security Audit

Report for CoboERC20Wrapper

Date: December 18, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Cobo Global
Target	CoboERC20Wrapper

Version History

Version	Date	Description
1.0	December 18, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-12-18 13:23:32+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 Target Contracts

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

The target of this audit is the code repository [1] of CoboERC20Wrapper of Cobo Global.

CoboERC20Wrapper is an upgradeable ERC-20 wrapper contract designed to represent deposits of an underlying ERC-20 token on a 1:1 basis. It uses OpenZeppelin upgradeable patterns (UUPS) and extensive role-based access control to manage minting, wrapping, pausing, upgrading, salvaging, and administration. Users with the WRAPPER_ROLE can deposit underlying tokens to mint wrapped tokens or burn wrapped tokens to withdraw the underlying asset. The contract enforces access and block lists, supports pausing, and preserves token decimals from the underlying asset. Additional safety features include controlled upgrades, token salvage restrictions, multicall support, and future-proofed storage gaps.

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

- evm/src/CoboERC20/CoboERC20Wrapper.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope 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
CoboERC20Wrapper	`Version 1`	`cc6e5a27c019901e0f2f4424597038c39f242727`
	`Version 2`	`b58a126be8f752f83c4443445681b365132208aa`

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.

[1] https://github.com/CoboGlobal/cobo-tokenization

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 smart contracts, i.e., the evaluation result does not 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 smart contracts.

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 smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts 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 have **four** recommendations and **four** notes.

- Recommendation: 4
- Note: 4

ID	Severity	Description	Category	Status
1	-	Non zero address checks	Recommendation	Fixed
2	-	Lack of non zero check of the parameter `value`	Recommendation	Partially Fixed
3	-	Comment typo	Recommendation	Fixed
4	-	Non-standard storage gap size	Recommendation	Fixed
5	-	Access control design for the function `deposit()` and `withdraw()`	Note	-
6	-	Design assumptions for underlying token	Note	-
7	-	Potential centralization risks	Note	-
8	-	Risk of privileged `burnFrom()` on wrapper's underlying token	Note	-

The details are provided in the following sections.

2.1 Recommendation

2.1.1 Non zero address checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `initialize()`, the address variable `underlyingToken` is not checked to ensure they are not zero. It is recommended to add such checks to prevent potential misoperations.

```
149   function initialize(
150       IERC20 underlyingToken,
151       string calldata name,
152       string calldata symbol,
153       string calldata uri,
154       address admin
155   ) external virtual initializer {
156       if (underlyingToken == this) {
157           revert ERC20InvalidUnderlying(address(this));
158       }
159       _underlying = underlyingToken;
160       _decimals = IERC20Metadata(address(_underlying)).decimals();
161
162       if (admin == address(0)) {
163           revert LibErrors.InvalidAddress();
164       }
165
```

```
166         __UUPSUpgradeable_init();
167         __ERC20_init(name, symbol);
168         __Multicall_init();
169         __Salvage_init();
170         __ContractUri_init(uri);
171         __Pause_init();
172         __RoleAccess_init();
173         __AccessList_init();
174
175         _grantRole(DEFAULT_ADMIN_ROLE, admin);
176     }
```

Listing 2.1: evm/src/CoboERC20/CoboERC20Wrapper.sol

Suggestion It is recommended to add such checks to prevent potential misoperations.

2.1.2 Lack of non zero check of the parameter `value`

Status Partially Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_recover()` does not check whether the `value` is non zero, which causes unnecessary operations to be executed when `value` equals zero.

```
448     function _recover(address account) internal virtual returns (uint256) {
449         uint256 value = IERC20(_underlying).balanceOf(address(this)) - totalSupply();
450         _mint(account, value);
451         return value;
452     }
```

Listing 2.2: evm/src/CoboERC20/CoboERC20Wrapper.sol

Similarly, the function `deposit()` and the function `withdraw()` also do not perform non zero checks on `value`.

```
302     function deposit(uint256 value) public virtual whenNotPaused onlyRole(WRAPPER_ROLE) returns (
            bool) {
303         address sender = _msgSender();
304         IERC20(_underlying).safeTransferFrom(sender, address(this), value);
305         _mint(sender, value);
306         emit Deposit(sender, value);
307         return true;
308     }
```

Listing 2.3: evm/src/CoboERC20/CoboERC20Wrapper.sol

```
313     function withdraw(uint256 value) public virtual whenNotPaused onlyRole(WRAPPER_ROLE) returns (
            bool) {
314         address sender = _msgSender();
315         _burn(sender, value);
316         IERC20(_underlying).safeTransfer(sender, value);
317         emit Withdrawal(sender, value);
318         return true;
319     }
```

Listing 2.4: evm/src/CoboERC20/CoboERC20Wrapper.sol

Suggestion It is recommended to add a check to ensure that `value` is greater than zero.

Clarification from BlockSec The function `_recover()` is fixed. For the function `deposit()` and the function `withdraw()`, the project claims that it does not require the verification logic of the value.

2.1.3 Comment typo

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The comments "CoboERC20" should be "CoboERC20Wrapper". To improve code readability, it is recommended to revise the following items:

```
136     * @notice This function configures the CoboERC20 contract with the initial state and granting
137     * privileged roles.
```

Listing 2.5: evm/src/CoboERC20/CoboERC20Wrapper.sol

```
185     * - {CoboERC20} is not paused.
```

Listing 2.6: evm/src/CoboERC20/CoboERC20Wrapper.sol

```
203     * - {CoboERC20} is not paused.
```

Listing 2.7: evm/src/CoboERC20/CoboERC20Wrapper.sol

```
230     * - {CoboERC20} is not paused.
```

Listing 2.8: evm/src/CoboERC20/CoboERC20Wrapper.sol

Suggestion Revise the logic accordingly.

2.1.4 Non-standard storage gap size

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract declares a storage gap to reserve space for future upgrades. However, since the contract declares state variables `_decimals` and `_underlying` which are packed into a single storage slot. Therefore, the size of `__gap` should not be 50 considering OpenZeppelin's convention.

Suggestion It's recommended to reduce the size of `__gap` to 49 to maintain the standard storage layout convention followed by OpenZeppelin.

2.2 Note

2.2.1 Access control design for the function `deposit()` and `withdraw()`

Introduced by `Version 1`

Description The contract `CoboERC20Wrapper` is designed as a wrapper that allows depositing underlying tokens to mint wrapped tokens and burning wrapped tokens to withdraw the underlying asset through the functions `deposit()` and `withdraw()`.

The functions `deposit()` and `withdraw()` implement access control exclusively through the `WRAPPER_ROLE` check. These functions operate correctly when called by accounts with the appropriate role.

The contract implements two access control approaches for different operations. The functions `mint()`, `transfer()`, and `transferFrom()` include the internal `_requireAccess()` check that validates account permissions against the protocol's whitelist and blacklist mechanisms for token holders.

```
313   function withdraw(uint256 value) public virtual whenNotPaused onlyRole(WRAPPER_ROLE) returns (
          bool) {
314       address sender = _msgSender();
315       _burn(sender, value);
316       IERC20(_underlying).safeTransfer(sender, value);
317       emit Withdrawal(sender, value);
318       return true;
319   }
```

Listing 2.9: evm/src/CoboERC20/CoboERC20Wrapper.sol

Feedback from the project The project team stated that this behavior is by design.

2.2.2 Design assumptions for underlying token

Introduced by `Version 1`

Description The contract `CoboERC20Wrapper` is designed to work with standard ERC20 tokens that maintain a 1:1 relationship between transferred amounts and wrapped token supply. The contract is not intended to support non-standard token implementations.

The function `deposit()` mints wrapped tokens based on the input `value` parameter, assuming the full amount is credited to the contract. The contract is not designed to handle tokens that deduct fees during transfer, as this would break the 1:1 mint ratio assumption.

```
302   function deposit(uint256 value) public virtual whenNotPaused onlyRole(WRAPPER_ROLE) returns (
          bool) {
303       address sender = _msgSender();
304       IERC20(_underlying).safeTransferFrom(sender, address(this), value);
305       _mint(sender, value);
306       emit Deposit(sender, value);
307       return true;
308   }
```

Listing 2.10: evm/src/CoboERC20/CoboERC20Wrapper.sol

The contract maintains a fixed wrapped token supply while the underlying token balance is expected to remain static. The contract is not designed to handle tokens with rebasing mechanisms where the balance changes automatically over time. While the function `_recover()` (invoked by `mint()`) exists to handle balance discrepancies, the core design assumes stable underlying balances for proper withdrawal operations.

```
444   /**
445    * @dev Mint wrapped token to cover any underlyingTokens that would have been transferred by
          mistake or acquired from
446    * rebasing mechanisms. Internal function that can be exposed with access control if desired.
447    */
448   function _recover(address account) internal virtual returns (uint256) {
449       uint256 value = IERC20(_underlying).balanceOf(address(this)) - totalSupply();
450       _mint(account, value);
451       return value;
452   }
```

Listing 2.11: evm/src/CoboERC20/CoboERC20Wrapper.sol

Feedback from the project The project team stated that `CoboERC20Wrapper` is a wrapped token designed to wrap `CoboERC20`, which follows standard ERC20 behavior. `CoboERC20` is an RWA token issued by the business entity and is not intended to circulate in the market in the short term, whereas `CoboERC20Wrapper` is allowed to circulate freely in the market.

2.2.3 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., `PAUSER_ROLE`) can conduct sensitive operations, which introduces potential centralization risks. For example, if the account of `PAUSER_ROLE` is maliciously manipulated, the attacker can pause the contract at any time, which will result in the freezing of contract deposit and withdrawal operations and affect the normal operation logic based on the protocol. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.2.4 Risk of privileged `burnFrom()` on wrapper's underlying token

Introduced by `Version 1`

Description In the contract `CoboERC20Wrapper`, if the underlying token is burnable (e.g, `CoboERC20`), special attention must be paid to the privileged function `burnFrom()`. Since the function `burnFrom()` allows authorized accounts to burn tokens from arbitrary addresses, it is important to ensure that the underlying token held by the wrapper contract is exempt from such actions. If the wrapper's underlying balance is burned, the wrapped tokens will become undercollateralized.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS