BLOCKSEC

Security Audit

Report for agent-contract

Date: September 3, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	olaxbt
Target	agent-contract

Version History

Version	Date	Description
1.0	September 3, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-09-03 11:32:10+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 agent-contract of olaxbt.

The protocol comprises two contracts: a Token contract and a Vesting contract. Upon deployment, the Token contract mints the specified amount directly to the Vesting contract. During initialization, the Vesting contract sets the beneficiary (token receiver) and the release schedule (start time and interval), and disburses tokens to the beneficiary according to the configured vesting logic.

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

- contracts/Token.sol
- contracts/Vesting.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
agent-contract	`Version 1`	`a228c3aec02f08de3d1670f3ed118c03d05b8c73`
	`Version 2`	`f960e48267e6ea4756ddae23ae68ebcc0295d0b4`

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 war-

[1] https://github.com/olaxbt/agent-contract

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

- High Risk: 1
- Medium Risk: 1
- Recommendation: 2
- Note: 1

ID	Severity	Description	Category	Status
1	High	Lock of tokens due to improper logic	Security Issue	Fixed
2	Medium	Lack of minted amount check in function `tgeRelease()`	Security Issue	Fixed
3	-	Fix typo	Recommendation	Fixed
4	-	Inconsistency between comments and code	Recommendation	Fixed
5	-	Potential centralization risk	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lock of tokens due to improper logic

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `vesting` contract defines fixed release amounts for each allocation. When the allocation cap is not evenly divisible by the release amount, the final release that would exceed the cap will revert. As a result, any remainder tokens from this indivisible portion remain permanently locked with no privileged function to recover the excess tokens.

```
 99   function release(Allocation allocation) external onlyAdmin {
100       AllocationInfo storage info = allocations[allocation];
101       require(info.nextRelease <= block.timestamp, "Release not yet");
102       require(
103           info.minted + info.releaseAmount <= info.cap,
104           "Release amount exceeds cap"
105       );
106       info.minted += info.releaseAmount;
107       info.nextRelease = block.timestamp + info.vestingPeriod;
108       token.transfer(info.recipient, info.releaseAmount);
109       emit Release(allocation, info.releaseAmount, info.nextRelease);
110   }
```

Listing 2.1: contracts/Vesting.sol

Impact Part of the token supply becomes inaccessible, breaking the intended distribution and potentially affecting tokenomics.

Suggestion Introduce a final release logic that adjusts the last transfer amount to fit the `cap`, or provide an `admin` function to recover any unreleased remainder.

2.1.2 Lack of minted amount check in function `tgeRelease()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `tgeRelease()`, the minted amount for the `Ecosystem` allocation is set to a fixed value without checking if any tokens were already released through the function `release()`. If prior releases occurred, this overwriting may cause the total minted tokens to exceed the allocation cap.

```
112   function tgeRelease() external onlyAdmin {
113       require(
114           allocations[Allocation.Liquidity].minted == 0,
115           "Already released liquidity"
116       );
117       require(
118           allocations[Allocation.Treasury].minted == 0,
119           "Already released treasury"
120       );
121       require(
122           allocations[Allocation.Marketing].minted == 0,
123           "Already released liquidity"
124       );
125       // release all liquidity
126       allocations[Allocation.Liquidity].minted = allocations[
127           Allocation.Liquidity
128       ].cap;
129       token.transfer(
130           allocations[Allocation.Liquidity].recipient,
131           allocations[Allocation.Liquidity].cap
132       );
133       // relase 5% marketing
134       allocations[Allocation.Marketing].minted = 2_500_000 ether;
135       token.transfer(
136           allocations[Allocation.Marketing].recipient,
137           allocations[Allocation.Marketing].minted
138       );
139       // release 1% treasury
140       allocations[Allocation.Treasury].minted = 10_000_000 ether;
141       token.transfer(
142           allocations[Allocation.Treasury].recipient,
143           allocations[Allocation.Treasury].minted
144       );
145       // release 6% Ecosystem
146       allocations[Allocation.EcoSystem].minted = 67_750_000 ether;
```

```
147         token.transfer(
148             allocations[Allocation.EcoSystem].recipient,
149             allocations[Allocation.EcoSystem].minted
150         );
151     }
```

Listing 2.2: contracts/Vesting.sol

Impact The minted tokens can exceed the intended allocation and disrupting tokenomics.

Suggestion Revise the logic accordingly.

2.2 Recommendation

2.2.1 Fix typo

Status Fixed in `Verison 2`

Introduced by `Version 1`

Description In the function `tgeRelease()`, it checks whether minted for the `Liquidity`, `Treasury`, and `Marketing` roles is zero. If so, it releases tokens according to the specified ratio. However, in the `Marketing` branch when `Marketing.minted != 0`, the error message should be "`Already released marketing`" rather than "`Already released liquidity`".

```
112     function tgeRelease() external onlyAdmin {
113         require(
114             allocations[Allocation.Liquidity].minted == 0,
115             "Already released liquidity"
116         );
117         require(
118             allocations[Allocation.Treasury].minted == 0,
119             "Already released treasury"
120         );
121         require(
122             allocations[Allocation.Marketing].minted == 0,
123             "Already released liquidity"
124         );
125         // release all liquidity
126         allocations[Allocation.Liquidity].minted = allocations[
127             Allocation.Liquidity
128         ].cap;
129         token.transfer(
130             allocations[Allocation.Liquidity].recipient,
131             allocations[Allocation.Liquidity].cap
132         );
133         // relase 5% marketing
134         allocations[Allocation.Marketing].minted = 2_500_000 ether;
135         token.transfer(
136             allocations[Allocation.Marketing].recipient,
137             allocations[Allocation.Marketing].minted
138         );
139         // release 1% treasury
```

```
140         allocations[Allocation.Treasury].minted = 10_000_000 ether;
141         token.transfer(
142             allocations[Allocation.Treasury].recipient,
143             allocations[Allocation.Treasury].minted
144         );
145         // release 6% Ecosystem
146         allocations[Allocation.EcoSystem].minted = 67_750_000 ether;
147         token.transfer(
148             allocations[Allocation.EcoSystem].recipient,
149             allocations[Allocation.EcoSystem].minted
150         );
151     }
```

Listing 2.3: contracts/Vesting.sol

Suggestion Revise the typos accordingly.

2.2.2 Inconsistency between comments and code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `tgeRelease()`, the comments describing the `Treasury` and `Ecosystem` token releases do not match the actual amounts transferred in the code. The discrepancy between documented percentages and implemented logic introduces ambiguity and may mislead auditors, developers, or users reviewing the contract.

```
112     function tgeRelease() external onlyAdmin {
113         require(
114             allocations[Allocation.Liquidity].minted == 0,
115             "Already released liquidity"
116         );
117         require(
118             allocations[Allocation.Treasury].minted == 0,
119             "Already released treasury"
120         );
121         require(
122             allocations[Allocation.Marketing].minted == 0,
123             "Already released liquidity"
124         );
125         // release all liquidity
126         allocations[Allocation.Liquidity].minted = allocations[
127             Allocation.Liquidity
128         ].cap;
129         token.transfer(
130             allocations[Allocation.Liquidity].recipient,
131             allocations[Allocation.Liquidity].cap
132         );
133         // relase 5% marketing
134         allocations[Allocation.Marketing].minted = 2_500_000 ether;
135         token.transfer(
136             allocations[Allocation.Marketing].recipient,
137             allocations[Allocation.Marketing].minted
```

```
138         );
139         // release 1% treasury
140         allocations[Allocation.Treasury].minted = 10_000_000 ether;
141         token.transfer(
142             allocations[Allocation.Treasury].recipient,
143             allocations[Allocation.Treasury].minted
144         );
145         // release 6% Ecosystem
146         allocations[Allocation.EcoSystem].minted = 67_750_000 ether;
147         token.transfer(
148             allocations[Allocation.EcoSystem].recipient,
149             allocations[Allocation.EcoSystem].minted
150         );
151     }
```

Listing 2.4: contracts/Vesting.sol

Suggestion Revise either the comments or implementation accordingly.

2.3 Note

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description In this project, `admin` can conduct sensitive operations, which introduces potential centralization risks. The function `release()` is only callable by the `admin`, that means, even after the unlock time has passed, if the `admin` does not call function `release()` in time, the receiver cannot receive the corresponding token. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS