BLOCKSEC

Security Audit

Report for Avalon USDAMinter

Date: January 20, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Avalon
Target	Avalon USDAMinter

Version History

Version	Date	Description
1.0	January 20, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-01-22 07:28:56+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 of Avalon USDAMinter [1], an implementation of the USDA token minter for Avalon.

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.

Project	Version	Commit Hash
Avalon USDAMinter	Version 1	3e0a468b429eb28e855748a36afb3ffd61cf3dfc

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 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.

[1] https://github.com/avalonfinancexyz/USDAMinter/tree/main

- **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 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 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 four 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.
- **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 **one** potential security issue. Besides, we have **one** recommendation and **two** notes.

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

ID	Severity	Description	Category	Status
1	Low	Potential DoS due to exhaustion of minting capability	DeFi Security	Confirmed
2	-	Add validation checks for setting functions to avoid redundant setting	Recommendation	Confirmed
3	-	Potential centralization risks	Note	-
4	-	Inconsistent record of the variable `user.redeemableAmount`	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential DoS due to exhaustion of minting capability

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `USDAMinter` contract, a global variable `mintedUSDACap` is used to limit the total amount of `USDA` tokens that can be minted. However, this design allows a user to consume the entire minting capability of the contract by minting the maximum amount of `USDA` tokens. As a result, this could prevent other users from minting `USDA` tokens, potentially leading to a Denial of Service (DoS) issue.

```
130   function mintUSDA(
131       uint256 desiredUsdaAmount
132   ) external whenNotPaused onlyWhitelisted {
133       require(desiredUsdaAmount > 0, "Amount must be greater than 0");
134
135       (uint256 requiredStableAmount, uint256 feeAmount) = previewMintUSDA(
136           desiredUsdaAmount
137       );
138
139       require(
140           mintedUSDA + desiredUsdaAmount <= mintedUSDACap,
141           "Exceeds mint cap"
142       );
```

Listing 2.1: src/USDAMinter.sol

Impact A user can consume the minting capability of the contract by minting the maximum amount of `USDA` tokens, which could lead to a DoS issue.

Suggestion Revise the code logic accordingly.

Feedback from the project This is our intended business design, and `mintedUSDACap` is a total cap, available on a first-come, first-served basis.

2.2 Additional Recommendation

2.2.1 Add validation checks for setting functions to avoid redundant setting

Status Confirmed

Introduced by `Version 1`

Description In the `USDAMinter` contract, the functions `setTreasury()`, `setFeeAddress()`, and `setWhitelist()` allow the contract admin or manager to assign values without any validation. It is recommended to add validation checks to these functions to avoid redundant settings.

```
227   function setTreasury(
228       address newTreasury
229   ) external onlyRole(DEFAULT_ADMIN_ROLE) {
230       require(newTreasury != address(0), "Invalid treasury address");
231       address oldTreasury = treasury;
232       treasury = newTreasury;
233       emit TreasuryUpdated(oldTreasury, newTreasury);
234   }
235
236   function setFeeAddress(
237       address newFeeAddress
238   ) external onlyRole(DEFAULT_ADMIN_ROLE) {
239       require(newFeeAddress != address(0), "Invalid fee address");
240       address oldFeeAddress = feeAddress;
241       feeAddress = newFeeAddress;
242       emit FeeAddressUpdated(oldFeeAddress, newFeeAddress);
243   }
```

Listing 2.2: src/USDAMinter.sol

```
254   function setWhitelist(address user) external onlyRole(MANAGER_ROLE) {
255       userInfo[user].isWhitelisted = true;
256       emit WhitelistUpdated(user, true);
257   }
```

Listing 2.3: src/USDAMinter.sol

Impact N/A

Suggestion Add validation checks to these functions to prevent redundant settings.

Feedback from the project We consider it a non-essential optimization with little impact, so we will not modify it.

2.3 Note

2.3.1 Potential centralization risks

Introduced by Version 1

Description The Avalon USDAMinter assigns different roles for various privileged operations, which introduces potential centralization risks. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.2 Inconsistent record of the variable `user.redeemableAmount`

Introduced by Version 1

Description In the redeeming process, the `redeemUSDA()` function burns `usdaAmount` of `USDA` tokens from users and sends the corresponding amount of stablecoins, based on the variable `actualUsdaAmount` (i.e., `usdaAmount - feeAmount`). However, the variable `user.redeemableAmount` is updated with `actualUsdaAmount`, leading to an inconsistent record. The `feeAmount` is not deducted from `user.redeemableAmount`, meaning it remains unredeemable.

```
168   function redeemUSDA(
169       uint256 usdaAmount
170   ) external whenNotPaused onlyWhitelisted {
171       require(usdaAmount > 0, "Amount must be greater than 0");
172
173       UserInfo storage user = userInfo[msg.sender];
174       uint256 feeAmount = (usdaAmount * user.fee) / FEE_DENOMINATOR;
175       uint256 actualUsdaAmount = usdaAmount - feeAmount;
176       uint256 stableAmount = convertToStable(actualUsdaAmount);
177       require(
178           user.redeemableAmount >= actualUsdaAmount,
179           "Exceeds minted amount"
180       );
181
182       // Burn USDA tokens
183       USDA.burn(msg.sender, usdaAmount);
184       if (feeAmount > 0) {
185           USDA.mint(feeAddress, feeAmount);
186       }
187
188       // Update state
189       mintedUSDA -= actualUsdaAmount;
190       user.redeemableAmount -= actualUsdaAmount;
```

Listing 2.4: src/USDAMinter.sol


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS