



Security Audit

Report for Avalon

Contracts

Date: October 11, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction	1
1.1 About Target Contracts	1
1.2 Disclaimer	1
1.3 Procedure of Auditing	2
1.3.1 Software Security	2
1.3.2 DeFi Security	2
1.3.3 NFT Security	3
1.3.4 Additional Recommendation	3
1.4 Security Model	3
Chapter 2 Findings	5
2.1 Software Security	5
2.1.1 Precision loss in the redeemUSDT function	5
2.2 DeFi Security	6
2.2.1 Invalid maximum borrow amount check	6
2.3 Additional Recommendation	8
2.3.1 Avoid multiplication following division to prevent precision loss	8
2.3.2 Avoid redundant state updates	8
2.3.3 Avoid using the deprecated library function	9
2.3.4 Ensure that addresses are not zero in constructors	9
2.4 Note	10
2.4.1 Potential centralization risks	10

Report Manifest

Item	Description
Client	Avalon
Target	Avalon Contracts

Version History

Version	Date	Description
1.0	October 11, 2024	First release

Signature

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

Avalon is a set of contracts that issues the [USDA](#) stablecoin, which is pegged 1:1 to the stablecoin [USDT](#). Borrowers can supply [FBTC](#) tokens to the [PoolManager](#) contract as collateral to mint [USDA](#) tokens, while lenders can provide [USDT](#) tokens in the [LendingPool](#) contract to benefit from a fixed reward rate.

The audit scope includes smart contracts from the following three repositories:

- [USDA](#) ¹: Contains the implementation of the [USDA](#) token.
- [AALoan](#) ²: Contains the [PoolManager](#) contract.
- [USDAPool](#) ³: Contains the [LendingPool](#) for Avalon.

Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of 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.

Project	Version	Commit Hash
AALoan	Version 1	18928e521f0e84d5602e5911841d228870b1d08e
	Version 2	13f633c5dbccd57c514ad7c3b594e58aad59beaf
USDA	Version 1	71d69f9a966bd9d594bf7046a2adfea53ad629a2
USDAPool	Version 1	2d94d64e1fdd66e9b078965c0ba970f7bbc9e458
	Version 2	65d3fb8fcl1a54d2ce2a517b48596a576d5a05c46

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/avalonfinancexyz/USDA>

²<https://github.com/avalonfinancexyz/AALoan/tree/v2.0>

³<https://github.com/avalonfinancexyz/USDAPool>

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 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 ⁴ and Common Weakness Enumeration ⁵. 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	High	High	Medium
	Low	Medium	Low
		High	Low
		Likelihood	

Accordingly, the severity measured in this report are classified into three categories: **High**,

⁴https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

⁵<https://cwe.mitre.org/>

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.

Chapter 2 Findings

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

- High Risk: 1
- Medium Risk: 1
- Recommendation: 4
- Note: 1

ID	Severity	Description	Category	Status
1	Medium	Precision loss in the <code>redeemUSDT</code> function	Software Security	Fixed
2	High	Invalid maximum borrow amount check	DeFi Security	Fixed
3	-	Avoid multiplication following division to prevent potential precision loss	Recommendation	Fixed
4	-	Avoid redundant state updates	Recommendation	Fixed
5	-	Avoid using the deprecated library function	Recommendation	Fixed
6	-	Ensure that addresses are not zero in constructors	Recommendation	Fixed
7	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Precision loss in the `redeemUSDT` function

Severity Medium

Status Fixed in [Version 2](#) (USDAPool)

Introduced by [Version 1](#) (USDAPool)

Description In the `redeemUSDT` function of the `LendingPool` contract, there is a precision loss issue in calculating the shares redeemable by the user. Specifically, `totalDepositedUSDT` can exceed `totalDepositedUSDTShares` because the interest on deposited `USDT` tokens accumulates in `totalDepositedUSDT`, while `totalDepositedUSDTShares` only increases when users deposit `USDT` tokens.

This precision loss could allow the `LENDER_ROLE` to manipulate the `_amount`, resulting in `redeemShares` being zero and enabling them to redeem `USDT` without reducing their shares. However, the rate of increase for `totalDepositedUSDT` is slow enough that executing such attacks would be too costly to yield a profit.

```
131 function redeemUSDT(  
132     uint256 _amount  
133 ) external onlyRole(LENDER_ROLE) realizeInterests {  
134     uint256 redeemShares = (_amount * totalDepositedUSDTShares) /  
135         totalDepositedUSDT;
```



```
136     require(
137         usdtDepositedShares[msg.sender] >= redeemShares,
138         "redeem amount exceeds balance"
139     );
140     usdtDepositedShares[msg.sender] -= redeemShares;
141     totalDepositedUSDTSshares -= redeemShares;
142     totalDepositedUSDT -= _amount;
143     USDT.safeTransfer(msg.sender, _amount);
144     emit RedeemUSDT(msg.sender, redeemShares, _amount);
145 }
```

Listing 2.1: contracts/LendingPool.sol

Impact Malicious actors can redeem a small amount of **USDT** tokens without decreasing shares.

Suggestion Use ceiling division, or require that redeeming shares are not zero.

2.2 DeFi Security

2.2.1 Invalid maximum borrow amount check

Severity High

Status Fixed in [Version 2](#) (AALoan)

Introduced by [Version 1](#) (AALoan)

Description In the [PoolManager](#) contract, users deposit **FBTC** token collateral to borrow **USDA** tokens. The protocol employs an over-collateralization mechanism, requiring that the maximum borrow amount does not exceed the collateral value multiplied by the [loanToValue](#) configuration variable.

```
550 function calculateMaxBorrowAmount(
551     uint256 loanToValue,
552     uint256 collateral,
553     uint256 debt,
554     uint256 FBTCOPrice,
555     uint256 USDADecimal,
556     uint256 FBTCODecimal,
557     uint256 oracleDecimal
558 ) public view returns (uint256) {
559     return
560         (((collateral * FBTCOPrice * 10 ** USDADecimal) /
561            (10 ** FBTCODecimal * 10 ** oracleDecimal)) * loanToValue) /
562         DENOMINATOR -
563         debt;
564 }
```

Listing 2.2: AALoan/src/protocol/PoolManager.sol

Additionally, the [PoolManager](#) provides a liquidation mechanism specifying that once the collateral value multiplied by the [liquidationThreshold](#) configuration variable is less than the debt value, the user becomes liquidatable. The [loanToValue](#) variable is set lower than the

`liquidationThreshold` to ensure the protocol maintains a buffer, preventing users from being liquidated immediately after borrowing.

```
599 function checkLiquidateCondition(  
600     uint256 liquidationThreshold,  
601     uint256 collateral,  
602     uint256 debt,  
603     uint256 FBTCOPrice,  
604     uint256 USDADecimal,  
605     uint256 FBTCODecimal,  
606     uint256 oracleDecimal  
607 ) public view returns (bool) {  
608     return  
609         collateral <  
610             (debt *  
611                 10 ** (oracleDecimal + FBTCODecimal - USDADecimal) *  
612                 DENOMINATOR) /  
613                 (FBTCOPrice * liquidationThreshold)  
614             ? true  
615             : false;  
616 }
```

Listing 2.3: AALoan/src/protocol/PoolManager.sol

However, users can still withdraw their collateral to get to reach the liquidation threshold. The withdrawal progress only requires that the user will not be liquidatable after the withdrawal. As a result, users can easily bypass the maximum borrow amount check.

```
572 function calculateMaxWithdrawAmount(  
573     uint256 liquidationThreshold,  
574     uint256 collateral,  
575     uint256 debt,  
576     uint256 FBTCOPrice,  
577     uint256 USDADecimal,  
578     uint256 FBTCODecimal,  
579     uint256 oracleDecimal  
580 ) public view returns (uint256) {  
581     if (debt == 0) {  
582         return collateral;  
583     } else {  
584         return  
585             collateral >  
586                 (debt *  
587                     10 ** (oracleDecimal + FBTCODecimal - USDADecimal) *  
588                     DENOMINATOR) /  
589                     (FBTCOPrice * liquidationThreshold)  
590             ? collateral -  
591                 (debt *  
592                     10 ** (oracleDecimal + FBTCODecimal - USDADecimal) *  
593                     DENOMINATOR) /  
594                     (FBTCOPrice * liquidationThreshold)  
595             : 0;  
596     }  
597 }
```

Listing 2.4: AALoan/src/protocol/PoolManager.sol

Impact Users can reach the liquidation line by withdrawing from the protocol, rendering the maximum borrow amount check invalid.

Suggestion Refactor the logic of the `calculateMaxWithdrawAmount` function.

2.3 Additional Recommendation

2.3.1 Avoid multiplication following division to prevent precision loss

Status Fixed in [Version 2](#) (USDAPool)

Introduced by [Version 1](#) (USDAPool)

Description In the `realizeInterests` function, there is an issue with multiplication occurring after division, which can amplify the precision loss from the division. It is recommended to switch the order of calculations to avoid this precision loss.

```

229 modifier realizeInterests() {
230     if (totalBorrowedUSDT != 0) {
231         uint256 totalInterest = ((apr * totalBorrowedUSDT) /
232             365 days /
233             APR_COEFFICIENT) * (block.timestamp - lastCheckpoint);
234         totalBorrowedUSDT += totalInterest;
235         totalDepositedUSDT += totalInterest;
236         unpaidInterestUSDT += totalInterest;
237     }
238     lastCheckpoint = block.timestamp;
239     _;
240 }
```

Listing 2.5: USDAPool/contracts/LendingPool.sol

Impact This may lead to unexpected results.

Suggestion Switch the calculation order to avoid precision loss.

2.3.2 Avoid redundant state updates

Status Fixed in [Version 2](#) (USDAPool)

Introduced by [Version 1](#) (USDAPool)

Description The `realizeInterests` function updates the state regardless of whether it is already the latest. Adding a check for `lastCheckpoint == block.timestamp` could help determine whether to perform the calculation and state update, thereby optimizing gas consumption.

```

229 modifier realizeInterests() {
230     if (totalBorrowedUSDT != 0) {
231         uint256 totalInterest = ((apr * totalBorrowedUSDT) /
232             365 days /
233             APR_COEFFICIENT) * (block.timestamp - lastCheckpoint);
```

```
234         totalBorrowedUSDT += totalInterest;
235         totalDepositedUSDT += totalInterest;
236         unpaidInterestUSDT += totalInterest;
237     }
238     lastCheckpoint = block.timestamp;
239     _;
240 }
```

Listing 2.6: USDAPool/contracts/LendingPool.sol

Impact N/A

Suggestion Add checks to prevent unnecessary state updates.

2.3.3 Avoid using the deprecated library function

Status Fixed in [Version 2](#) (USDAPool)

Introduced by [Version 1](#) (USDAPool)

Description The code uses an outdated version of the OpenZeppelin library that employs the deprecated function `_setupRole`.

```
56     constructor(address _admin, address _usda, address _usdt) {
57         require(_admin != address(0), "_!_admin");
58         _setupRole(DEFAULT_ADMIN_ROLE, _admin);
59         _setRoleAdmin(ADMIN_ROLE, ADMIN_ROLE);
60         _setRoleAdmin(MANAGER_ROLE, ADMIN_ROLE);
61         _setRoleAdmin(LENDER_ROLE, ADMIN_ROLE);
62         _setRoleAdmin(BORROWER_ROLE, ADMIN_ROLE);
63
64         _setupRole(ADMIN_ROLE, _admin);
65         _setupRole(MANAGER_ROLE, _admin);
66
67         USDA = IERC20(_usda);
68         USDT = IERC20(_usdt);
69     }
```

Listing 2.7: USDAPool/contracts/LendingPool.sol

Impact N/A

Suggestion Avoid using the deprecated function.

2.3.4 Ensure that addresses are not zero in constructors

Status Fixed in [Version 2](#) (USDAPool)

Introduced by [Version 1](#) (USDAPool)

Description When setting critical address parameters, it is recommended to check that the addresses are not zero to avoid misconfiguration.

```
56     constructor(address _admin, address _usda, address _usdt) {
57         require(_admin != address(0), "_!_admin");
58         _setupRole(DEFAULT_ADMIN_ROLE, _admin);
```

```
59     _setRoleAdmin(ADMIN_ROLE, ADMIN_ROLE);
60     _setRoleAdmin(MANAGER_ROLE, ADMIN_ROLE);
61     _setRoleAdmin(LENDER_ROLE, ADMIN_ROLE);
62     _setRoleAdmin(BORROWER_ROLE, ADMIN_ROLE);
63
64     _setupRole(ADMIN_ROLE, _admin);
65     _setupRole(MANAGER_ROLE, _admin);
66
67     USDA = IERC20(_usda);
68     USDT = IERC20(_usdt);
69 }
```

Listing 2.8: USDAPool/contracts/LendingPool.sol

Impact The lack of non-zero address checks may lead to misconfiguration.

Suggestion Add proper checks.

2.4 Note

2.4.1 Potential centralization risks

Introduced by [Version 1](#)

Description Avalon Contracts include several critical functions, such as upgrading contracts, setting key parameters, and modifying user states, which can only be executed by privileged roles. Misconfigurations of these parameters can significantly impact the functionality of the contracts, potentially rendering them unusable. Consequently, if the private keys of these privileged roles were compromised, the entire protocol could be incapacitated, leading to potential centralization risks.

