Security Audit Report for Avalon Contracts

Date: October 30, 2024 **Version:** 1.3

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Avalon
Target	Avalon Contracts

Version History

Version	Date	Description
1.0	October 11, 2024	First release
1.1	October 22, 2024	Add two new repositories
1.2	October 25, 2024	Refactor interest distribution
1.3	October 30, 2024	Update `SavingAccount` to properly support the OFT token

Signature

Digitally signed by: BlockSec (https://blocksec.com)
SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C
Date: 2024-10-30 12:42:29+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

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.
- USDA-OFT [2]: Contains the USDa and the USDaOFTAdapter for Avalon. After Version 1, a new contract, sUSDa, was introduced during the audit and was used by the SavingAccount contract.
- AALoan [3]: Contains the PoolManager contract.
- USDAPool [4]: Contains the LendingPool for Avalon.
- Saving-account [5]: Contains the SavingAccount for Avalon. This contract will only be deployed on Ethereum.

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.

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/USDa-oft

[3] https://github.com/avalonfinancexyz/AALoan/tree/v2.0

[4] https://github.com/avalonfinancexyz/USDAPool

[5] https://github.com/avalonfinancexyz/saving-account

Project	Version	Commit Hash
AALoan	Version 1	18928e521f0e84d5602e5911841d228870b1d08e
	Version 2	13f633c5dbccd57c514ad7c3b594e58aad59beaf
USDA	Version 1	71d69f9a966bd9d594bf7046a2adfea53ad629a2
USDAPool	Version 1	2d94d64e1fdd66e9b078965c0ba970f7bbc9e458
	Version 2	65d3fb8fc1a54d2ce2a517b48596a576d5a05c46
Saving-account	Version 1	3806530473d4ec9775c2664577c6744f9109b734
	Version 2	f513db33ab1d669fdd66dfe365003ebb6233f91b
	Version 3	623bea02da6d6d5885dac66d84b1fc8690bf9598
	Version 4	a5651e2e65f4c78384ee4133097f77f38abd166d
USDA-OFT	Version 1	393349344296c03bf4d5512c8ede7daf2b83cf24
	Version 2	15ee68086919de24c6d39d27b84bb5e7f55a160b

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 [6] and Common Weakness Enumeration [7]. 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.

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

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

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.

Chapter 2 Findings

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

- High Risk: 1
- Medium Risk: 1
- Low Risk: 2
- Recommendation: 6
- Note: 1

ID	Severity	Description	Category	Status
1	Medium	Precision loss in the `redeemUSDT` function	Software Security	Fixed
2	Low	Inconsistent usage and calculation of the underlying asset	Software Security	Fixed
3	High	Invalid maximum borrow amount check	DeFi Security	Fixed
4	Low	Potential risks in interest distribution	DeFi Security	Fixed
5	-	Avoid multiplication following division to prevent precision loss	Recommendation	Fixed
6	-	Avoid redundant state updates	Recommendation	Fixed
7	-	Avoid using the deprecated library function	Recommendation	Fixed
8	-	Ensure that addresses are not zero in constructors	Recommendation	Fixed
9	-	Remove unused variables	Recommendation	Acknowledged
10	-	Remove the redundant check	Recommendation	Fixed
11	-	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.

```
function redeemUSDT(
    uint256 _amount
) external onlyRole(LENDER_ROLE) realizeInterests {
    uint256 redeemShares = (_amount * totalDepositedUSDTShares) /
        totalDepositedUSDT;
    require(
        usdtDepositedShares[msg.sender] >= redeemShares,
        "redeem amount exceeds balance"
    );
    usdtDepositedShares[msg.sender] -= redeemShares;
    totalDepositedUSDTShares -= redeemShares;
    totalDepositedUSDT -= _amount;
    USDT.safeTransfer(msg.sender, _amount);
    emit RedeemUSDT(msg.sender, redeemShares, _amount);
}
```

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.

2.1.2 Inconsistent usage and calculation of the underlying asset

Severity Low

Status Fixed in `Version 2` (Saving-account)

Introduced by `Version 1` (Saving-account)

Description In the `SavingAccount` contract, there are two functions, `getSharesByAmount` and `getAmountByShares`, that allow users to calculate the conversion ratio between shares and underlying tokens. However, the minting and redeeming processes are inconsistent with these two functions.

Specifically, users deposit tokens to earn interest in this contract, with the interest accumulated in `unpaidInterest` and distributed when the admin invokes the `distributeInterests` function. The two calculation functions treat `unpaidInterest` as part of the underlying asset, while the minting and redeeming processes do not, leading to inconsistency.

```
function distributeInterests()
    external
    realizeReward
    onlyRole(POOL_MANAGER_ROLE)
{
    IERC20(usda).safeTransferFrom(
        msg.sender,
        address(this),
        unpaidInterest
    );
    totalUnderlying += unpaidInterest;
    unpaidInterest = 0;
```

```
}
```

Listing 2.2: saving-account/contracts/SavingAccount.sol

```
function distributeInterests()
    external
    realizeReward
    onlyRole(POOL_MANAGER_ROLE)
{
    IERC20(usda).safeTransferFrom(
        msg.sender,
        address(this),
        unpaidInterest
    );
    totalUnderlying += unpaidInterest;
    unpaidInterest = 0;
}
```

Listing 2.3: saving-account/contracts/SavingAccount.sol

```
function getTotalUnderlying() public view returns (uint256) {
    // need include manager fee
    uint256 totalInterest = getRPS() * (block.timestamp - lastCheckpoint);
    return totalUnderlying + totalInterest;
}

/**
 * @dev get amount of shares by underlying amount
 * @param _amount the amount of underlying
 */
function getSharesByAmount(uint256 _amount) public view returns (uint256) {
    uint256 sTokenTotalSupply = susda.totalSupply();
    if (sTokenTotalSupply == 0) {
        return 0;
    } else {
        return (_amount * sTokenTotalSupply) / getTotalUnderlying();
    }
}

/**
 * @dev get amount by shares
 * @param _shares the amount of cToken
 */
function getAmountByShares(uint256 _shares) public view returns (uint256) {
    uint256 sTokenTotalSupply = susda.totalSupply();
    if (sTokenTotalSupply == 0) {
        return 0;
    } else {
        return (_shares * getTotalUnderlying()) / (sTokenTotalSupply);
    }
}
```

Listing 2.4: saving-account/contracts/SavingAccount.sol

```
function _mintFor(
    uint256 amount,
    address receiver
) internal realizeReward nonReentrant whenNotPaused{
    usda.safeTransferFrom(msg.sender, address(this), amount);

    uint256 cTokenAmount;
    uint256 sTokenTotalSupply = susda.totalSupply();
    if (sTokenTotalSupply == 0 || totalUnderlying == 0) {
        cTokenAmount = amount;
    } else {
        cTokenAmount = (amount * sTokenTotalSupply) / totalUnderlying;
    }

    susda.mint(receiver, cTokenAmount);

    totalUnderlying = totalUnderlying + amount;
}

/**
 * @dev redeem susda
 * @param amount the amount of sToken, 1 sToken = 10**18, which eaquals to 1 usda (if not interest).
 * error code
 * 100: redeem is less than the balance
 * 101: totalUnderlying should be greater than 0
 * 102:
 */
function redeem(uint256 amount) external realizeReward nonReentrant whenNotPaused{
    require(amount <= susda.balanceOf(msg.sender), "less than balance");
    require(totalUnderlying > 0, "totalUnderlying = 0");
    require(susda.totalSupply() > 0, "susda total is 0");
    require(amount >= 1e18, "redeem amount is too small");

    uint256 sTokenTotalSupply = susda.totalSupply();

    uint256 underlyingAmount = (amount * totalUnderlying) /
        sTokenTotalSupply;

    susda.burn(msg.sender, amount);

    totalUnderlying = totalUnderlying - underlyingAmount;

    redeemIndex++;
    redeemDetails[redeemIndex] = RedeemDetail({
        id: redeemIndex,
        timestamp: block.timestamp,
        user: msg.sender,
        underlyingAmount: underlyingAmount,
        isDone: false
    });

```

```
    emit RedeemRequested(
        redeemIndex,
        block.timestamp,
        msg.sender,
        underlyingAmount
    );
}
```

Listing 2.5: saving-account/contracts/SavingAccount.sol

Impact Inconsistent usage may lead to unexpected miscalculations for users of the view functions.

Suggestion Refactor the code accordingly.

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.

```
function calculateMaxBorrowAmount(
    uint256 loanToValue,
    uint256 collateral,
    uint256 debt,
    uint256 FBTC0Price,
    uint256 USDADecimal,
    uint256 FBTC0Decimal,
    uint256 oracleDecimal
) public view returns (uint256) {
    return
        (((collateral * FBTC0Price * 10 ** USDADecimal) /
            (10 ** FBTC0Decimal * 10 ** oracleDecimal)) * loanToValue) /
        DENOMINATOR -
        debt;
}
```

Listing 2.6: 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.

```
function checkLiquidateCondition(
    uint256 liquidationThreshold,
    uint256 collateral,
    uint256 debt,
    uint256 FBTC0Price,
    uint256 USDADecimal,
    uint256 FBTC0Decimal,
    uint256 oracleDecimal
) public view returns (bool) {
    return
        collateral <
            (debt *
                10 ** (oracleDecimal + FBTC0Decimal - USDADecimal) *
                DENOMINATOR) /
                (FBTC0Price * liquidationThreshold)
            ? true
            : false;
}
```

Listing 2.7: 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.

```
function calculateMaxWithdrawAmount(
    uint256 liquidationThreshold,
    uint256 collateral,
    uint256 debt,
    uint256 FBTC0Price,
    uint256 USDADecimal,
    uint256 FBTC0Decimal,
    uint256 oracleDecimal
) public view returns (uint256) {
    if (debt == 0) {
        return collateral;
    } else {
        return
            collateral >
                (debt *
                    10 ** (oracleDecimal + FBTC0Decimal - USDADecimal) *
                    DENOMINATOR) /
                    (FBTC0Price * liquidationThreshold)
                ? collateral -
                    (debt *
                        10 ** (oracleDecimal + FBTC0Decimal - USDADecimal) *
                        DENOMINATOR) /
                    (FBTC0Price * liquidationThreshold)
                : 0;
    }
}
```

Listing 2.8: 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.

2.2.2 Potential risks in interest distribution

Severity Low

Status Fixed in `Version 3` (Saving-account)

Introduced by `Version 1` (Saving-account)

Description In the `distributeInterests` function of the `SavingAccount` contract, interests are periodically transferred into the contract and added to `totalUnderlying` by the admin. If a user deposits just before this distribution, the user may receive rewards disproportionate to the actual deposit.

```
function distributeInterests()
    external
    realizeReward
    onlyRole(POOL_MANAGER_ROLE)
{
    IERC20(usda).safeTransferFrom(
        msg.sender,
        address(this),
        unpaidInterest
    );
    totalUnderlying += unpaidInterest;
    unpaidInterest = 0;
}
```

Listing 2.9: saving-account/contracts/SavingAccount.sol

Impact Users may receive rewards that are not proportionate to their actual deposits.

Suggestion Refactor the code logic to ensure accurate interest distribution.

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.

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

```
            totalBorrowedUSDT += totalInterest;
            totalDepositedUSDT += totalInterest;
            unpaidInterestUSDT += totalInterest;
        }
        lastCheckpoint = block.timestamp;
        _;
    }
```

Listing 2.10: 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.

```
    modifier realizeInterests() {
        if (totalBorrowedUSDT != 0) {
            uint256 totalInterest = ((apr * totalBorrowedUSDT) /
                365 days /
                APR_COEFFICIENT) * (block.timestamp - lastCheckpoint);
            totalBorrowedUSDT += totalInterest;
            totalDepositedUSDT += totalInterest;
            unpaidInterestUSDT += totalInterest;
        }
        lastCheckpoint = block.timestamp;
        _;
    }
```

Listing 2.11: 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`.

```
    constructor(address _admin, address _usda, address _usdt) {
        require(_admin != address(0), "!_admin");
        _setupRole(DEFAULT_ADMIN_ROLE, _admin);
        _setRoleAdmin(ADMIN_ROLE, ADMIN_ROLE);
```

```
    _setRoleAdmin(MANAGER_ROLE, ADMIN_ROLE);
    _setRoleAdmin(LENDER_ROLE, ADMIN_ROLE);
    _setRoleAdmin(BORROWER_ROLE, ADMIN_ROLE);

    _setupRole(ADMIN_ROLE, _admin);
    _setupRole(MANAGER_ROLE, _admin);

    USDA = IERC20(_usda);
    USDT = IERC20(_usdt);
}
```

Listing 2.12: 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.

```
constructor(address _admin, address _usda, address _usdt) {
    require(_admin != address(0), "!_admin");
    _setupRole(DEFAULT_ADMIN_ROLE, _admin);
    _setRoleAdmin(ADMIN_ROLE, ADMIN_ROLE);
    _setRoleAdmin(MANAGER_ROLE, ADMIN_ROLE);
    _setRoleAdmin(LENDER_ROLE, ADMIN_ROLE);
    _setRoleAdmin(BORROWER_ROLE, ADMIN_ROLE);

    _setupRole(ADMIN_ROLE, _admin);
    _setupRole(MANAGER_ROLE, _admin);

    USDA = IERC20(_usda);
    USDT = IERC20(_usdt);
}
```

Listing 2.13: USDAPool/contracts/LendingPool.sol

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

Suggestion Add proper checks.

2.3.5 Remove unused variables

Status Acknowledged

Introduced by `Version 1` (USDA-OFT)

Description In the `USDa` contract, the `mintVault` and `burnVault` variables are not used.

Impact N/A

Suggestion Remove the unused variables.

2.3.6 Remove the redundant check

Status Fixed in `Version 2` (Saving-account)

Introduced by `Version 1` (Saving-account)

Description When users redeem their shares in the `SavingAccount` contract, the contract checks that the redeeming amount is less than the user's `susda` balance. However, since the burning process of `susda` already verifies this condition, the check is redundant.

```
function redeem(uint256 amount) external realizeReward nonReentrant whenNotPaused{
    require(amount <= susda.balanceOf(msg.sender), "less than balance");
    require(totalUnderlying > 0, "totalUnderlying = 0");
    require(susda.totalSupply() > 0, "susda total is 0");
    require(amount >= 1e18, "redeem amount is too small");

    uint256 sTokenTotalSupply = susda.totalSupply();

    uint256 underlyingAmount = (amount * totalUnderlying) /
        sTokenTotalSupply;

    susda.burn(msg.sender, amount);
```

Listing 2.14: saving-account/contracts/SavingAccount.sol

Impact N/A

Suggestion Remove the redundant check.

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.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS