Security Audit Report for IntentAssets Contracts

Date: June 7, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	DAppOS
Target	IntentAssets Contracts

Version History

Version	Date	Description
1.0	June 7, 2024	First Release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-06-07 12:27:11+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 focus of this audit is on the IntentAssets Contracts of the DAppOS protocol [1]. These contracts facilitate the transition between different assets and support cross-chain transfers.

It is important to note that only the contracts located within the `contracts` folder in the repository are included in the scope of this audit. 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.

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
IntentAssets Contracts	Version 1	c9b770eb1076976ad60e037d513f0e5423834765
	Version 2	fdbdca39d782cf2059c4751d710cb38b169cebdc

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] https://github.com/DappOSDao/IntentAssets/tree/main

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 [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 **three** potential security issues. Besides, we have **one** recommendation and **six** notes.

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

ID	Severity	Description	Category	Status
1	Low	Inconsistent parameter between invocation and interface	Software Security	Fixed
2	Medium	Potential precision loss	DeFi Security	Fixed
3	Medium	Lack of validation on oracle feeds	DeFi Security	Fixed
4	-	Require `msg.value` to be 0 when processing ERC-20 tokens	Recommendation	Acknowledged
5	-	Potential centralization risks	Note	-
6	-	Potential arbitrage risks	Note	-
7	-	Potential DoS on `CachePool` redemption	Note	-
8	-	Ensure sufficient funds are kept for `CachePool` redemption	Note	-
9	-	Assumptions regarding the `WearChecker` contract	Note	-
10	-	The design of circulate in the `MainPool`	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Inconsistent parameter between invocation and interface

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `processPluginTransaction` function of the `CachePool` contract, the call to the `wearChecker.checkWear` is inconsistent with the interface. Specifically, the `inputToken` is put as the second parameter in the invocation, but it should be the first parameter according to the interface.

```
function processPluginTransaction(
    address tokenToReceive,
    address inputToken,
    uint256 inputAmount,
    IExecutionPlugin pluginContract,
    bytes memory data
) external onlyRole(TRANSFER_ROLE) nonReentrant whenNotPaused {
    require(plugins[pluginContract], "CachePool: Unauthorized plugin.");
```

```
        uint256 balanceBefore = Common._getTokenBalance(tokenToReceive);

        TransferHelper.safeTransfer2(
            inputToken,
            address(pluginContract),
            inputAmount
        );
        pluginContract.execute(tokenToReceive, inputToken, inputAmount, data);

        uint256 receivedAmount = Common._getTokenBalance(tokenToReceive) -
            balanceBefore;
        require(
            wearChecker.checkWear(
                tokenToReceive,
                inputToken,
                receivedAmount,
                inputAmount
            ),
            "CachePool: Excessive wear."
        );
        emit PluginExecuted(
            tokenToReceive,
            inputToken,
            inputAmount,
            receivedAmount,
            pluginContract
        );
    }
```

Listing 2.1: contracts/core/pools/cachePool/CachePool.sol

```
interface IWearChecker {
function checkWear(
    address tokenIn,
    address tokenOut,
    uint256 receivedAmount,
    uint256 amountOut
) external returns (bool);
}
```

Listing 2.2: contracts/core/pools/cachePool/IWearChecker.sol

Impact The inconsistency may lead to unexpected behaviors.

Suggestion Unify the order of parameters.

2.2 DeFi Security

2.2.1 Potential precision loss

Severity Medium

Status Fixed in `Version 2`

Introduced by Version 1

Description In the MainPool contract, precision loss issues exist in both the redeem and circulate functions. When calculating the IntentTokens to burn in the redeem function, the result may be rounded down to zero (line 153), allowing users to withdraw underlying assets without burning any IntentTokens.

```
140  function redeem(address token, uint256 tokenAmount) external nonReentrant {
141      MintData storage mintDataStorage = mintData[msg.sender][token];
142      uint256 totalUnderlyingAssetAmount = mintDataStorage
143          .totalUnderlyingAssetAmount;
144      uint256 totalMinted = mintDataStorage.totalMinted;
145      require(
146          tokenAmount <= totalUnderlyingAssetAmount,
147          "MainPool: Insufficient balance."
148      );
149
150      if (totalMinted != 0) {
151          uint256 _intentTokenToRedeem = (totalUnderlyingAssetAmount == 0)
152              ? totalMinted
153              : (tokenAmount * totalMinted) / totalUnderlyingAssetAmount;
154
155          mintDataStorage.totalMinted -= _intentTokenToRedeem;
156          intentToken.burnFrom(msg.sender, _intentTokenToRedeem);
157
158          emit Redeemed(
159              msg.sender,
160              token,
161              address(intentToken),
162              tokenAmount,
163              _intentTokenToRedeem
164          );
165      }
166      totalDeposited[token] -= tokenAmount;
167      mintDataStorage.totalUnderlyingAssetAmount -= tokenAmount;
168      TransferHelper.safeTransfer2(token, msg.sender, tokenAmount);
169
170      emit Withdrawn(msg.sender, token, tokenAmount);
171  }
```

Listing 2.3: contracts/core/pools/mainPool/MainPool.sol

Similarly, the result may be rounded down to zero when calculating the circulated underlying assets in the circulate function (line 210). Consequently, totalMinted is reduced from the user's mintData while totalUnderlyingAmount remains unchanged. In the most extreme case, totalMinted can be reduced to zero while totalUnderlyingAmount retains a non-zero value, allowing users to withdraw underlying assets without burning IntentTokens.

```
179  function circulate(
180      address user,
181      uint256 intentTokenAmount
182  ) external onlyIntentToken whenIsNotCollectingStats returns (bool) {
183      uint256 totalCirculatedAmount = 0;
```

```
        uint256 underlyingAssetsLength = underlyingAssets.length();
        for (uint256 i = 0; i < underlyingAssetsLength; ++i) {
            MintData storage mintDataStorage = mintData[user][
                underlyingAssets.at(i)
            ];
            uint256 totalUnderlyingAssetAmount = mintDataStorage
                .totalUnderlyingAssetAmount;
            uint256 totalMinted = mintDataStorage.totalMinted;
            uint256 circulatableAmount = Calculator
                .getIntentTokenAmountByUnderlyingAsset(
                    underlyingAssets.at(i),
                    totalUnderlyingAssetAmount,
                    intentToken,
                    intentToken.priceOracle()
                );

            if (circulatableAmount == 0) continue;
            uint256 amountToCirculate = totalMinted;

            if (amountToCirculate > intentTokenAmount - totalCirculatedAmount) {
                amountToCirculate = intentTokenAmount - totalCirculatedAmount;
            }
            if (amountToCirculate > circulatableAmount) {
                amountToCirculate = circulatableAmount;
            }

            uint256 underlyingAssetsToUse = (totalUnderlyingAssetAmount *
                amountToCirculate) / circulatableAmount;

            totalDeposited[underlyingAssets.at(i)] -= underlyingAssetsToUse;
            (
                mintDataStorage.totalMinted,
                mintDataStorage.totalUnderlyingAssetAmount
            ) = (
                totalMinted - amountToCirculate,
                totalUnderlyingAssetAmount - underlyingAssetsToUse
            );
            totalCirculatedAmount += amountToCirculate;

            if (totalCirculatedAmount == intentTokenAmount) {
                return true;
            }
        }
        return false;
    }
```

Listing 2.4: contracts/core/pools/mainPool/MainPool.sol

Impact Users may withdraw underlying assets from the `MainPool` without burning `IntentTokens`, causing protocol loss.

Suggestion Revise the logic accordingly.

2.2.2 Lack of validation on oracle feeds

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description The PriceOracle contract fetches prices from Chainlink-like oracles. However, it doesn't verify if the fetched price is within a valid range or is not a stale value.

```
function _getAssetPriceInBase(
    address tokenAddress
) internal view returns (uint256) {
    address feed = chainlinkFeeds[tokenAddress];
    if (feed != address(0)) {
        (, int price, , , ) = AggregatorV3Interface(feed).latestRoundData(); // using Chainlink
            price feed method
        return uint256(price);
    } else {
        ICustomizedOracle customizedOracle = customizedOracles[
            tokenAddress
        ];
        require(
            address(customizedOracle) != address(0),
            "PriceOracle: Unsupported asset."
        );
        return customizedOracle.getPrice();
    }
}
```

Listing 2.5: contracts/core/utils/oracle/PriceOracle.sol

Impact Incorrect or stale prices may be used as the latest price to convert between assets, leading to inaccurate asset valuations.

Suggestion Check the validity and freshness of the price data.

2.3 Additional Recommendation

2.3.1 Require msg.value to be 0 when processing ERC-20 tokens

Status Acknowledged

Introduced by Version 1

Description The submit function in the MainPool contract allows users to deposit either native or ERC-20 tokens. It is recommended to add a check to ensure that msg.value is zero when users deposit ERC-20 tokens.

```
function submit(
    address token,
    uint256 tokenAmount
) external payable whenNotPaused whenIsNotCollectingStats nonReentrant {
    DepositConfig memory _depositConfig = depositConfig[token];
```

```
        require(
            _depositConfig.isSupported &&
                tokenAmount >= _depositConfig.minDepositAmount,
                "MainPool:: not qualified"
        );

        if (token == address(0)) {
            require(
                msg.value == tokenAmount,
                "MainPool: Ether amount mismatch."
            );
        } else {
            TransferHelper.safeTransferFrom(
                token,
                msg.sender,
                address(this),
                tokenAmount
            );
        }
```

Listing 2.6: contracts/core/pools/mainPool/MainPool.sol

Impact The input native tokens may be locked in the contract.

Suggestion Add a check on `msg.value`.

2.4 Note

2.4.1 Potential centralization risks

Introduced by `Version 1`

Description The protocol introduces several roles that can set key parameters within the smart contract system. For example, the `UPDATE_BASE_ASSET_RATE_ROLE` can set the `baseAssetRate` parameter, representing the exchange rate between `IntentTokens` and the corresponding base tokens. Exploitation of these privileged roles can result in an incorrect state of the entire smart contract system.

Feedback from the Project

1. For `DEFAULT_ADMIN_ROLE`, it is an OpenZeppelin `Timelock` contract, with the executor and scheduler being a Gnosis Safe multi-sig wallet.
2. For `UPDATE_BASE_ASSET_RATE_ROLE`, it is also a contract that has rate change ratio limits and rate update internal limits (which are beyond the scope of this audit).

We will implement these contracts and conduct a re-audit in due course. In any case, we will strive to decentralize these contracts and impose numerical limits on any risky operations.

2.4.2 Potential arbitrage risks

Introduced by `Version 1`

Description The protocol uses Chainlink price feeds to convert between underlying assets and base assets, and a `baseAssetRate` to convert between base assets and `IntentTokens`. In the `IntentToken` contract, the `updateBaseAssetRate` function allows the `UPDATE_BASE_ASSET_RATE_ROLE` to modify the `baseAssetRate`. However, this introduces potential arbitrage risks. An attacker can initiate a sandwich attack to profit from changes in the Chainlink price or `baseAssetRate`. For instance, a malicious user can sandwich an `updateBaseAssetRate` transaction with their own `submit` and `redeem` transactions. If the `baseAssetRate` is increased in the `updateBaseAssetRate` transaction, the user can withdraw more underlying assets than initially deposited, exploiting the rate change.

Feedback from the Project Our quotaChecker will, to some extent, make arbitrage infeasible:

1. White list checks on `msg.sender` are required for both `submit` and `redeem`.
2. Checks regarding quotas, frequencies, etc., remain in place for submission and redemption.
3. We will minimize the time gap between two updates as much as possible, greatly limiting the arbitrage opportunities within a single update.

In conclusion, we enforce white list verification for arbitrageurs and adjust the timing of rate updates, along with the quotas for each submission and redemption, making it impossible for arbitrageurs to profitably offset gas costs.

2.4.3 Potential DoS on `CachePool` redemption

Introduced by `Version 1`

Description The protocol contains two types of pools that allow users to mint or redeem `IntentTokens`: `MainPool` and `CachePool` (via the `IntentTokenMinting` contract). The `MainPool`'s `redeem` function requires users to record the corresponding `mintData` (i.e., by depositing the corresponding underlying asset), preventing users from redeeming `IntentTokens` that are minted via the `IntentTokenMinting` contract. Conversely, `IntentTokens` minted in the `MainPool` can be redeemed for underlying assets in both the `MainPool` and `CachePool`.

Attackers can initiate a Denial-of-Service (DoS) attack as follows:

1. Mint an excessive amount of `IntentTokens` in the `MainPool`.
2. Redeem these `IntentTokens` to drain the underlying assets in the `CachePool`.

Consequently, users cannot redeem if their `IntentTokens` are minted via the `IntentTokenMinting` contract, resulting in a DoS.

Feedback from the Project When a user mints in the `MainPool`, circulates, and finally redeems in the `CachePool`, potentially initiating a DoS attack, we can resolve the issue by transferring the circulated funds from the `MainPool` to the `CachePool` using the `transferToWhitelist` method in the `MainPool`.

2.4.4 Ensure sufficient funds are kept for `CachePool` redemption

Introduced by `Version 1`

Description The `TRANSFER_ROLE` of `CachePool` is privileged to call the `transferToMainPool` function to transfer arbitrary tokens to the `MainPool`. However, the function doesn't check if there are any pending redemption requests nor if there are enough funds for users to claim all pending requests. If the `CachePool` contract is left without enough funds to cover the requests, users may be unable to redeem their tokens. Therefore, the protocol should be aware of such risks and ensure that the `CachePool` contract maintains sufficient funds for redemption.

```
function transferToWhitelist(
    address token,
    address to,
    uint256 amount
) external onlyRole(TRANSFER_ROLE) nonReentrant whenNotPaused {
    require(whitelisted[to], "MainPool: Not whitelisted.");
    TransferHelper.safeTransfer2(token, to, amount);
    require(
        Common._getTokenBalance(token) >= totalDeposited[token],
        "MainPool: Insufficient balance."
    );
    emit TransferToWhitelist(token, to, amount);
}
```

Listing 2.7: contracts/core/pools/mainPool/MainPool.sol

Feedback from the Project We make sure all the redemption in `CachePool` can be done by `MainPool.transferToWhitelist(address(cachePool))`.

2.4.5 Assumptions regarding the `WearChecker` contract

Introduced by `Version 1`

Description The `CachePool` contract relies on a `WearChecker` contract to perform checks after the plugin contract execution. According to the context, the `WearChecker` contract should check the slippage of swapping input tokens to output tokens. In common cases, oracles are required for such checks. However, the input or output tokens are not limited to ensure corresponding oracles are supported. Besides, the `WearChecker` contract should check whether the `tokenToReceive` is valid.

It should be noted that the `WearChecker` contract implementation is not included in the audited code repository and falls outside of the audit scope. For the purposes of this audit, it is assumed that the validation logic within the `WearChecker` contract is correct.

Feedback from the Project

1. `tokenToUse` and `tokenToReceive` will be limited in `WearChecker`.
2. `tokenToReceive` will be checked if it is valid.
3. If no price info returns from oracle, it will fail.

2.4.6 The design of circulate in the `MainPool`

Introduced by `Version 1`

Description For the `IntentTokens` minted through the `MainPool` contract, when the tokens are transferred or burnt, there is a circulate logic to reduce the underlying assets of the token sender according to the proportion of token transferred out.

However, in the `circulate` function, the reduction of the user's underlying asset is not proportional (as a common design pattern used by other protocols). Instead, the reduction happens in an order set by the `configureDeposits` function. In the case of violent price fluctuation (specifically, large price movement difference between underlying assets), it can lead to losses for the users.

Feedback from the Project

1. When the prices of underlying assets fluctuate sharply, the risks associated with circulate are equivalent to those associated with submit through the intentTokenMinting contract. Users need to be aware of these risks before using these functions.
2. In the event of severe exchange rate fluctuations, we may temporarily suspend circulate to ensure user safety.
3. For LSD, LRT tokens, price retrieval by the oracle tends to favor direct reading from the pool to mitigate short-term volatility caused by DEX and similar platforms.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS