Aria Protocol Security Review

Pashov Audit Group

Conducted by: eeyore, Shaka, shaflow, IvanFitro, p_tsanev, Madalad

April 25th 2025 - April 28nd 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **AriaProtocol/main-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Aria Protocol

Aria Protocol is a platform that enables fractional ownership of intellectual property by crowdsourcing funding, tokenizing IP assets into ERC20 tokens, and distributing royalties to stakeholders. It uses Vaults and disctribution contract to let users invest in IP, claim fractionalized tokens, and earn rewards.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 17c554e901f6c741003d74ae72db4fb2ef9662df

fixes review commit hash - d6060ae83ca2c4a25bc277f7a4a8ccbdfc87c7f8

Scope

The following smart contracts were in scope of the audit:

- `AriaIPVault`
- `AriaIPVaultStorage`
- `IAriaIPVault`
- `VaultAssetRegistryAdmin`
- `VaultFundraiseAdmin`
- `VaultWhitelistAdmin`
- `IVaultAdmin`
- `IVaultAdmin`
- `IVaultFundraiseAdmin`
- `VaultAdmin`
- `AriaIPVaultFactory`
- `IAriaIPVaultFactory`
- `IVaultFundraiseUser`
- `VaultFundraise`
- `VaultFundraiseStorage`
- `VaultStateChecker`
- `IVaultView`
- `VaultView`
- `Whitelist`
- `WhitelistStorage`

7. Executive Summary

Over the course of the security review, eeyore, Shaka, shaflow, IvanFitro, p_tsanev, Madalad engaged with Aria to review Aria Protocol. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Aria Protocol
Repository	https://github.com/AriaProtocol/main-contracts
Date	April 25th 2025 - April 28nd 2025
Protocol Type	Fundraising and shared ownership

Findings Count

Severity	**Amount**
Medium	1
Low	11
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Incorrect integration	Medium	Resolved
[L-01]	Inconsistent storage location	Low	Resolved
[L-02]	Unnecessary UUPS upgradeable logic	Low	Resolved
[L-03]	Malicious admin could block users from withdrawing tokens	Low	Resolved
[L-04]	DoS on updateUsdcContractAddress() by 1 wei deposit	Low	Resolved
[L-05]	VaultFundraiseAdmin.withdraw() reverts if the vault has multiple tokens	Low	Resolved
[L-06]	Users in VaultFundraise cannot cancel deposits	Low	Acknowledged
[L-07]	Users need admin trust for fractional IP in fundraise	Low	Resolved
[L-08]	Flawed mechanism for fractional token withdrawal	Low	Resolved
[L-09]	Fundraise admin has the power to deny all depositors their fractional token	Low	Acknowledged
[L-10]	Function can be reverted for publicly mintable SPG NFTs	Low	Acknowledged
[L-11]	fractionalTokenTotalSupply can be less than the totalDeposits	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] Incorrect integration

Severity

Impact: Medium

Likelihood: Medium

Description

The `_registerIpAndAttachTermsAndCollectRoyaltyTokens()` function incorrectly integrates with the updated `mintAndRegisterIpAndAttachPILTermsAndDistributeRoyaltyTokens()` function from Story Protocol.

link

The updated version of `mintAndRegisterIpAndAttachPILTermsAndDistributeRoyaltyTokens()` now requires an array of `WorkflowStructs.LicenseTermsData[]` as input and returns a `uint256[] memory licenseTermsIds`. However, the current integration expects a single `WorkflowStructs.LicenseTermsData` and a single `uint256 licenseTermsId`, leading to type mismatches.

Specifically:

- `licenseTermsData` should be an array (`LicenseTermsData[]`), but is passed as a single struct.
- `licenseTermsId` should be an array (`uint256[]`), but is treated as a single `uint256` in the return.

This mismatch leads to a DoS and the inability to register and fractionalize IP.

Recommendations

- Update `_registerIpAndAttachTermsAndCollectRoyaltyTokens()` to pass an array of `LicenseTermsData[]` to the `mintAndRegisterIpAndAttachPILTermsAndDistributeRoyaltyTokens()` function.
- Adjust the return type of `licenseTermsId` to handle an array (`uint256[]`), matching the updated function's return value.

Fix

PR 20: story_protocol integration

8.2. Low Findings

[L-01] Inconsistent storage location

The `VaultFundraiseStorage` and `WhitelistStorage` contracts do not follow the correct convention for calculating namespace-based storage locations (ERC-7201), leading to inconsistencies between the contracts.

- In `VaultFundraiseStorage`, the formula is not followed, and the `bytes32` value is not pre-calculated.
- Similarly, in `WhitelistStorage`, the storage location is not calculated using the expected formula.

To resolve this, ensure that the correct formula for calculating namespace-based storage locations is followed consistently across `VaultFundraiseStorage` and `WhitelistStorage`.

Fix:

PR 21: erc7201 storage location for Fundraise & Whitelist

[L-02] Unnecessary UUPS upgradeable logic

The `AriaIPVault` contract inherits from `UUPSUpgradeable` and implements `_authorizeUpgrade`, but this is redundant given the contract use of a `BeaconProxy` deployed via the `AriaIPVaultFactory` contract, which points to a `UpgradeableBeacon`.

In this architecture, the `BeaconProxy` points to the beacon, and upgrades should be handled via the beacon's `upgradeTo` method, not through the `UUPS` upgrade functionality. The presence of `UUPSUpgradeable` adds unnecessary complexity and could lead to confusion in how upgrades are managed.

Since `BeaconProxy` does not manage its own implementation and simply delegates to the beacon, the `_authorizeUpgrade` function will have **no effect**

on the implementation. The upgrade logic is controlled by the beacon, not the proxy.

To resolve this, consider removing the `UUPSUpgradeable` inheritance and the `_authorizeUpgrade` method from `AriaIPVault`, and instead use only `Initializable` to properly handle initialization.

Fix:

PR 32: use UUPS instead Beacon for AriaIPVault

[L-03] Malicious admin could block users from withdrawing tokens

```
function updateFractionalTokenTotalSupply(uint256 newTotalSupply)
        external
        override
        onlyRole(AccessControlStorage.DEFAULT_ADMIN_ROLE)
    {
        AriaIPVaultStorage.VaultLayout storage $ = AriaIPVaultStorage.load();


        if ($.vaultType == VaultType.Fundraise) {
            if
              (newTotalSupply < $$.totalDeposits[$$.setup.usdcContractAddress]) {


                                          newTotalSupply, $$.totalDeposits[$$.setup.usd
                );
            }
        }
        if ($.fractionalToken != address(0)) {
            revert Errors.AriaIPVault__FractionalTokenAlreadyDeployed
              ($.fractionalToken);
        }

        uint256 previousSupply = $.tokenDetails.fractionalTokenTotalSupply;
        $.tokenDetails.fractionalTokenTotalSupply = newTotalSupply;

        emit FractionalTokenTotalSupplyUpdated(
          {previousTotalSupply:previousSupply,
          newTotalSupply:newTotalSupply}
        );
    }
```

Allowing a malicious admin to increase the `FractionalTokenTotalSupply` to a very large value after fundraising is completed could cause an overflow when users try to claim their tokens. This overflow could lead to incorrect calculations or even a loss of tokens, potentially disrupting the entire token distribution system.

It is recommended to set an overall increase limit.

Fix:

PR 24: overflow when fractionalTokenTotalSupply is updated

[L-04] DoS on `updateUsdcContractAddress()` by 1 wei deposit

`VaultFundraiseAdmin.updateUsdcContractAddress()` allows the admin to update the USDC contract address for an open fundraise, but only if no deposits have been made yet.

A call to this function when the deposits are zero can be easily frontrun with a deposit of 1 wei, which will make the function revert.

In order to prevent this, it is recommended to implement one of the following solutions:

- Require a minimum deposit amount.
- Remove the revert condition, as the `totalDeposits` variable will account for deposits on both the old and new USDC contracts. In this case, it would also be necessary to adapt the `claimRefund()` and `_fundraiseCalculateClaim()` functions to account for the different USDC contract addresses.

Fix:

PR 30: account for many USDC used of fundraise lifetime

[L-05] `VaultFundraiseAdmin.withdraw()` reverts if the vault has multiple tokens

`VaultFundraiseAdmin.withdraw()` can be called by the admin to withdraw all USDC deposited in the vault after the fundraiser is over. The function iterates over all USDC addresses registered in the vault and transfers the balance to the fund receiver.

```
uint256 length = $.tokensInVault.length();

        tokens = new address[](length);
        withdrawnAmounts = new uint256[](length);
        for (uint256 i = 0; i < length; i++) {
                tokens[i] = $.tokensInVault.at(i);
                withdrawnAmounts[i] = IERC20(tokens[i]).balanceOf(address(this));
@>              $.tokensInVault.remove(tokens[i]);
@>              IERC20(tokens[i]).safeTransfer($.setup.fundReceiver, withdrawnAmounts[
        }
```

This implementation has two issues:

1. In each iteration, the current token is removed from the `tokensInVault` enumerable set, which will change the index of the other tokens, causing an "array out-of-bounds access" error, and locking all tokens in the vault. E.g., if there are two tokens in the vault, the token at index 0 is removed, leaving only one token in the set. The next iteration will try to access the token at index 1, which no longer exists.

2. In each iteration, the token is transferred to the fund receiver, so if the transfer fails, the rest of the tokens cannot be withdrawn.

The current implementation does not allow for more than one token to be added to the `tokensInVault` set. However given that the contract is upgradeable, this could change in the future.

It is recommended to either:

- Remove the `tokensInVault` variable and always use the `setup.usdcContractAddress` variable instead, or

- If in the future it is expected to have multiple tokens in the vault, receive the token address as a parameter in the `withdraw()` function so that each token can be withdrawn separately. In this case, it would also be necessary to adapt the `claimRefund()` and `_fundraiseCalculateClaim()` functions to account for the different USDC contract addresses.

Fix:

PR 30: account for many USDC used of fundraise lifetime

[L-06] Users in VaultFundraise cannot cancel deposits

```
function deposit
      (address erc20, uint256 amount) external virtual override nonReentrant {


        $._checkAndUpdateState();
        if (amount == 0) revert Errors.AriaIPVault__ZeroDepositAmount
          (msg.sender, erc20);
        if
          (erc20 != $.setup.usdcContractAddress) revert Errors.AriaIPVault__InvalidUSD
        if
          ($.state != FundraiseState.Open) revert Errors.AriaIPVault__VaultNotOpen($.s

        $.tokensInVault.add(erc20);
        $.deposits[msg.sender][erc20] += amount;
        $.totalDeposits[erc20] += amount;

        IERC20(erc20).safeTransferFrom(msg.sender, address(this), amount);

        emit DepositReceived
          ({depositor: msg.sender, token: erc20, amount: amount});
    }
```

In the VaultFundraise system, once a user chooses to deposit funds, they can only withdraw their funds by waiting for the admin to close or cancel the vault. However, during the fundraising process, the vault parameters, such as the total token supply, may change. Users are unable to cancel their deposit based on these changes, which could lead to issues if they want to withdraw their funds before the campaign ends or if there are unexpected changes to the fundraising parameters.

This lack of flexibility could be problematic, especially if the user wants to avoid potential losses due to changes in the vault's terms or conditions.

Recommendations:

Implementing a deposit cancellation function is recommended, allowing users to cancel deposits during the fundraising period.

[L-07] Users need admin trust for fractional IP in fundraise

For fundraising, the registration and fractionalizing of the IP is executed by the admin only after the fundraise if completed. This means that the participants of

the fundraise are expected to trust the admin to register and fractionalize the IP and do it with the correct data, as there is no possibility to recover the deposited funds even if the registration never happens.

Recommendations:

Given that the creation of vaults is permissionless, it would be recommended to require the admin to register the IP and fractionalize it at the creation of the fundraise. Otherwise, it would be recommended to explicitly state in the documentation that users should not deposit funds into the fundraise if they do not trust the admin to make them aware of the risks.

Also, FractionalToken can be properly set before allowing the admin to withdraw the funds.

```
function withdraw()
        external
        override
        onlyRole(AccessControlStorage.DEFAULT_ADMIN_ROLE)
        returns (address[] memory tokens, uint256[] memory withdrawnAmounts)
    {

+       AriaIPVaultStorage.VaultLayout storage $ = AriaIPVaultStorage.load();
+       if ($$.fractionalToken == address
+ (0)) revert Errors.AriaIPVault__FractionalTokenNotSet();
```

Fix:

PR 28: users trust for IP and factory admin only

[L-08] Flawed mechanism for fractional token withdrawal

`VaultAdmin` provides the admin with a two-step mechanism for withdrawing fractional tokens from the vault. This could be used to withdraw fractional tokens not claimed after a certain period of time, and the dust remainder caused by the rounding error in the calculation of the fractional tokens.

However, the `initFractionalTokenWithdrawal()` and `execFractionalTokenWithdrawal()` functions rely on the fractional tokens to be owned by the vault, which is not the case. The vault does not own any fractional tokens but is instead the minter of the fractional tokens. This means that in order to recover the fractional tokens, instead of transferring out the balance of the vault, it is necessary to mint the tokens.

Recommendations:

Change the implementation of the `initFractionalTokenWithdrawal()` and `execFractionalTokenWithdrawal()` functions to mint the fractional tokens instead of transferring them from the vault's balance.

Fix:

PR 25: flawed mechanism for admin fractional token withdrawal/mint

[L-09] Fundraise admin has the power to deny all depositors their fractional token

The fundraise admin is the only role, supposedly trusted by the system, tasked with executing crucial vault operations like fractionalization. He can also, with imposed coded limitations, change configurations like the state of the fundraise, addresses, timelocks, etc. What he should not be allowed to do, however, is grief depositors expecting fractional tokens. Due to the admin's power to seamlessly close the fundraise before expiration, via `closeRaise()` the following scenario is possible:

1. Take out a sufficiently large flashloan on the current deposit token
2. Make a huge deposit on his own behalf, inflating the total deposits
3. Close the fundraise via `closeRaise()`
4. Claim most, if not all depending on decimals, of the fractional tokens, leaving depositors with nothing
5. Reclaiming his deposit, alongside the deposits of honest users, via `VaultFindraiseAdmin#withdraw()` and repaying the flashloan

All of this can be executed in a single transaction. No admin should be able to tamper with the rewards of users. The simplest fix would be to introduce a short timelock for the `closeRaise()` and `cancelRaise()` functions as well, to make the attack non-atomic and disrupting the flashloan.

[L-10] Function can be reverted for publicly mintable SPG NFTs

The vault admin, through the registry, is able to fractionalize any SPG NFT, register it as an Intellectual Property (IP), attach terms and collect the maximum royalties, which are later distributed. This fractionalized NFT is exactly the token that is redeemed by the users of the vault, by either having deposited during fundraising, or being part of the whitelist. The function `_registerIpAndAttachTermsAndCollectRoyaltyTokens()` internally calls `mintAndRegisterIpAndAttachPILTermsAndDistributeRoyaltyTokens` on the Story's distribution workflow, setting `allowDuplicates` to false. The distribution workflow then calls `_mintAndRegisterIp` attempting to register the freshly minted NFT. This could be problematic as some SPG NFTs are publicly mintable:

```
function mint(
        address to,
        string calldata nftMetadataURI,
        bytes32 nftMetadataHash,
        bool allowDuplicates
    ) public virtual returns (uint256 tokenId) {
        //@audit if public minting is enabled, we can mint any metadata hash
        if (!_getSPGNFTStorage()._publicMinting && !hasRole
          (SPGNFTLib.MINTER_ROLE, msg.sender)) {
            revert Errors.SPGNFT__MintingDenied();
        }
        tokenId = _mintToken({
            to: to,
            payer: msg.sender,
            nftMetadataURI: nftMetadataURI,
            nftMetadataHash: nftMetadataHash,
            allowDuplicates: allowDuplicates
        });
    }
```

This, combined with the fact that Story's mempool is public, could allow any arbitrary user to directly mint the SPG with an identical metadata hash, causing a revert due to the disallowed duplicates:

```
tokenId = $._nftMetadataHashToTokenId[nftMetadataHash];
        if (!allowDuplicates && tokenId != 0) {
            revert Errors.SPGNFT__DuplicatedNFTMetadataHash({
                spgNftContract: address(this),
                tokenId: tokenId,
                nftMetadataHash: nftMetadataHash
            });
        }


        if ($._mintFeeToken != address(0) && $._mintFee > 0) {
            IERC20($._mintFeeToken).safeTransferFrom(payer, address
              (this), $._mintFee);
        }


        tokenId = ++$._totalSupply;
        if ($._nftMetadataHashToTokenId[nftMetadataHash] == 0) {
            // only store the token ID if the metadata hash is not used
            $._nftMetadataHashToTokenId[nftMetadataHash] = tokenId;
        }
```

Recommendations:

Apparently, unless the `VaultAssetRegistryAdmin` has a valid minter role, the only SPGs we can interact with are publicly mintable ones:

```
modifier onlyMintAuthorized(address spgNftContract) {
        if (
            !ISPGNFT(spgNftContract).hasRole
              (SPGNFTLib.MINTER_ROLE, msg.sender) &&
            !ISPGNFT(spgNftContract).publicMinting()
        ) revert Errors.Workflow__CallerNotAuthorizedToMint();
        _;
    }
}
```

thus, the best solution here would be to allow duplicates: `allowDuplicates: true`

[L-11] `fractionalTokenTotalSupply` can be less than the `totalDeposits`

When a fractional token is deployed within a Fundraise Vault, it is checked to ensure that the `fractionalTokenTotalSupply` is greater than the `totalDeposits`.

```
function _deployFractionalToken(
        address ipId,
        address fractionalTokenTemplate
    ) internal returns (address fractionalToken) {
        AriaIPVaultStorage.VaultLayout storage $ = AriaIPVaultStorage.load();


        if ($.vaultType == VaultType.Fundraise) {
@>          if
  ($.tokenDetails.fractionalTokenTotalSupply < $$.totalDeposits[$$.setup.usdcContractA

                    $.tokenDetails.fractionalTokenTotalSupply,
                    $$.totalDeposits[$$.setup.usdcContractAddress]
                );
            }
        }

      ///code...
    }
```

The problem is that it doesn't account for the possibility of different token decimals. For example, while the fractional token has 18 decimals (as it's a standard ERC20 token), USDC has only 6 decimals.

Since the decimals aren't properly scaled, it can lead to the `fractionalTokenTotalSupply` being less than the `totalDeposits`. This discrepancy can result in users receiving fewer fractional tokens than intended, causing a loss of funds for the users.

Let's use an example to better understand the problem. Assume the Fundraise Vault has finished, and the deposits are in USDC, with `totalDeposits = 10_000e6` and `fractionalTokenTotalSupply = 1e18`.

1. The fractional token is deployed.
2. The check `$.tokenDetails.fractionalTokenTotalSupply < $$.totalDeposits[$$.setup.usdcContractAddress]` is performed and passes because `fractionalTokenTotalSupply > totalDeposits`, i.e., `1e18 > 10_000e6`. However, this is incorrect because it compares tokens with different decimals.

The correct approach would be to scale the USDC amount. In this case, `10_000e6 * 1e12` (18 - 6) results in `10_000e18`. With this scaling, the check would fail because `10_000e18 > 1e18`.

Recommendations:

To resolve the issue, scale the token decimals to ensure a correct comparison.

Fix:

PR 26: fractional supply and total deposits compared on same decimals