Arsen Blockchain Security

After Protocol

Security review

Review was done by:

- Arsen
- Naman
- Koby

Table of content

Introduction

About Arsen Security

Arsen Security is a Web3 and blockchain security firm led by Arsen, a professional smart contract auditor with a proven track record on public Web3 hacking platforms. Our team is composed of top-tier auditors who consistently rank in bounty programs and security contests, demonstrating their expertise and commitment to securing the blockchain ecosystem.

Disclaimer

While we are committed to delivering top-tier Web3 security services, it is essential to recognize that no protocol can be guaranteed completely secure. The evolving nature of Web3 technologies means new vulnerabilities and threats may arise over time. As a result, we do not provide any explicit or implied warranties regarding the absolute security of reviewed protocols. We strongly advise clients to implement continuous security measures and monitoring to mitigate potential risks.

Risk classification

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

- Critical - Severe loss of funds, complete compromise of project availability, or significant violation of protocol invariants.
- High - High impact on funds, major disruption to project functionality or substantial violation of protocol invariants.
- Medium - Moderate risk affecting a noticeable portion of funds, project availability or partial violation of protocol invariants.
- Low - Low impact on a small portion of funds (e.g. dust amount), minor disruptions to service or minor violation of protocol invariants.
- Informational - Negligible risk with no direct impact on funds, availability or protocol invari- ants.

Executive Summary

After is a lending platform for Aerodrome and Velodrome veNFTs, enabling users to borrow against their veNFTs with no interest or liquidation risk while earning yield. Lenders benefit from boosted rewards on liquid AERO and VELO tokens, with risk managed through individual markets offering different loan-to-value ratios.

2.2 Summary

Project Name	After protocol
Project Type	Lending
Repository	After Finance
Commit Hash	c617f23
Review Period	02/01/25 - 10/01/25

2.3 Issues found

Severity	Count
Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	4
Informational	2

Findings

Medium Risk

[M-1] - The borrowing fee/fee receiver aren't set during market creation, which postpones the correct market usage.

Vulnerability Detail

The problem is that once the market is created via the factory, for some time the market can't function correctly. It happens because the setBorrowingFee and setFees aren't called during the market creation itself.

```
function createMarket(uint256 loanToValue) public onlyOwner returns (address) {
        if (loanToValue == 0 || loanToValue > 10000) revert InvalidLoanToValue();
        (bool exists,) = _markets.tryGet(loanToValue);
        if (exists) revert MarketExists();

        address market = address(new Market(owner(), share, staked, loanToken, loanToValue));
        IStaked(staked).grantRole(IStaked(staked).MARKET(), market);
        _markets.set(loanToValue, market);
        return market;
    }
```

Proof of Concept

1. Factory creates the market but doesn't set the borrowing fee and fees address.
2. User deposit money into the protocol
3. Once the borrower decide to borrow the money, he would face the revert.

Recommendation

Assign the fee address and BorrowingFee during the market creation in the factory.

```
function createMarket(uint256 loanToValue, uint256 _borrowingFee) public onlyOwner returns (address) {
        if (loanToValue == 0 || loanToValue > 10000) revert InvalidLoanToValue();
        (bool exists,) = _markets.tryGet(loanToValue);
        if (exists) revert MarketExists();

        address market = address(new Market(owner(), share, staked, loanToken, loanToValue));
        IStaked(staked).grantRole(IStaked(staked).MARKET(), market);
        _markets.set(loanToValue, market);

        market.setBorrowingFee(_borrowingFee); ++
        market.setFees(feeAddress); ++
        return market;
    }
```

Findings

Low Risk

[L-1] - safeTransfer must be used

Vulnerability Detail

The transfer() and transferFrom() method is used instead of safeTransfer() and safeTransferFrom(), presumably to save gas however OpenZeppelin's documentation discourages the use of transferFrom(), use safeTransferFrom() whenever possible because safeTransferFrom auto-handles boolean return values whenever there's an error.

```
function transfer(address to, address[] calldata tokens) public onlyRole(VOTER) {
        for (uint256 i = 0; i < tokens.length; i++) {
            uint256 balance = IERC20(tokens[i]).balanceOf(address(this));
            IERC20(tokens[i]).transfer(to, balance); //@audit must be used the safeTransfer
        }
    }
```

Using safeTransferFrom has the following benefits -

- It checks the boolean return values of ERC20 operations and reverts the transaction if they fail,
- at the same time allowing you to support some non-standard ERC20 tokens that don't have boolean return values.
- It additionally provides helpers to increase or decrease an allowance, to mitigate an attack possible with vanilla approve.

Recommendation

Consider using safeTransfer() and safeTransferFrom() instead of transfer() and transferFrom().

Findings

Low Risk

[L-2] - Due to the rounding, accumulated leftover amount can’t be claimed overtime.

Vulnerability Detail

In the staking.sol, due to rounding of the rewardRate during the notifyRewardAmount, some left over amount can accumulate over time. However, there is no method, which would allow the owner to claim/retrieve the left over amount.

```
function notifyRewardAmount(uint256 reward) external onlyOwner updateReward(address(0)) {

        if (block.timestamp ≥ periodFinish) {
            rewardRate = reward / rewardsDuration;
            //@audit
            //rewardRate = 100e18 / 604800 → 165343915343915.3439153439153439
        } else {
            uint256 remaining = periodFinish - block.timestamp;
            uint256 leftover = remaining * rewardRate;
            rewardRate = (reward + leftover) / rewardsDuration;
        }

        .......

        if (rewardRate * rewardsDuration > balance) revert RewardTooHigh();

          .......
        emit RewardAdded(reward);
    }
```

Recommendation

Create the method which would allow to claim the leftover amount resulted due to rounding.

[L-3] - setMinimumBalance must be checked

Vulnerability Detail

I would recommend to make the proper check related to the minimumBalance setting. Check like protection against the min/max value, so some unintended mistakes from the owner side can be avoided.

```
function setMinimumBalance(uint256 _minimumBalance) public onlyRole(DEFAULT_ADMIN_ROLE) {
        emit MinimumBalanceUpdate(minimumBalance, _minimumBalance);
        minimumBalance = _minimumBalance;
    }
```

Recommendation

```
function setMinimumBalance(uint256 _minimumBalance) public onlyRole(DEFAULT_ADMIN_ROLE) {
                require(_minimumBalance > 0, "0 value");
                require(_minimumBalance < type(uint256).max, "0 value");
        emit MinimumBalanceUpdate(minimumBalance, _minimumBalance);
        minimumBalance = _minimumBalance;
    }
```

Findings

Low Risk

[L-4] -Missing zero amount validation in Market.borrow() and Staked.marketStake()

Vulnerability Detail

While the main Staking contract validates against zero amounts, the Market contract's borrow() function and the Staked contract's marketStake() function lack zero amount validation. This oversight allows the creation of zero-value positions in the protocol.

Proof of Concept

The vulnerability exists in two connected contracts:

1. In Market.sol:

```
function borrow(address to, uint256 collateral) public virtual nonReentrant returns (uint256 receiveAmount) {
        uint256 borrowAmount = collateral * loanToValue / 10000; //LTV is in the basis point --> 1 token *
5000 / 10000 = 0.5 tokens
        uint256 availableLiquidity = IERC20(loanToken).balanceOf(address(this)); //check avaliable liquidity
        if (borrowAmount > availableLiquidity) revert InsufficientLiquidity(); //check there is enough
liquidity
```

From the code above, there is no check to verify that the collateral is not zero. 2. In Staked.sol:

```
function startStake(uint256 amount) public nonReentrant returns (uint256) {
        IERC20(share).transferFrom(msg.sender, address(this), amount);
        uint256 nextEpochAt = block.timestamp - (block.timestamp % 7 days) + 7 days;
        uint256 unlocksAt = nextEpochAt + 48 hours;
```

Recommendation

Add the check related to the 0 amount

Findings

Informational

[I-3] - Use ownable2Step

Vulnerability Detail
The "Ownable2Step" pattern is an improvement over the traditional "Ownable" pattern, designed to enhance the security of ownership transfer functionality in a smart contract. Unlike the original "Ownable" pattern, where ownership can be transferred directly to a specified address, the "Ownable2Step" pattern introduces an additional step in the ownership transfer process. Ownership transfer only completes when the proposed new owner explicitly accepts the ownership, mitigating the risk of accidental or unintended ownership transfers to mistyped addresses.

Impact
Without the "Ownable2Step" pattern, the contract owner might inadvertently transfer ownership to an unintended or mistyped address, potentially leading to a loss of control over the contract. By adopting the "Ownable2Step" pattern, the smart contract becomes more resilient against external attacks aimed at seizing ownership or manipulating the contract's behaviour.

Recommendation
It is recommended to use either Ownable2Step or Ownable2StepUpgradeable depending on the smart contract.

[I-1] - Floating and Outdated Pragma

Vulnerability Detail
Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities. The contract allowed floating or unlocked pragma to be used, i.e., ^0.8.0. This allows the contracts to be compiled with all the solidity compiler versions above the limit specified.

Impact
If the smart contract gets compiled and deployed with an older or too recent version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions or unidentified exploits in the new versions. Incompatibility issues may also arise if the contract code does not support features in other compiler versions, therefore, breaking the logic.The likelihood of exploitation is low.

Recommendation
Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.28 pragma version
Reference: https://swcregistry.io/docs/SWC-103
Additionally, here is a powerful tool which reveals the current solidity compiler bugs: https://00xsev.github.io/solidityBugsByVersion/