PRISM SEC
End-To-End Blockchain Security Solutions
Auditing, Penetration Testing,
Adversary Simulation, AI Security Testing, & More

Pinlink Security Review

By Prism

Auditors

Lead Security Researcher: Jacobo Lansac (@Jacopod)

July 5, 2025

Contents

1 Introduction **2**

2 About Prism **2**

3 About Pinlink **2**

4 Disclaimer **2**

5 Risk classification **2**

5.1 Impact . . . 2

5.2 Likelihood . . . 2

5.3 Action required for severity levels . . . 3

6 Executive Summary **3**

6.1 Files in scope . . . 3

6.2 Architecture review . . . 3

6.2.1 System overview . . . 3

6.2.2 Architecture assessment . . . 4

6.2.3 User flow . . . 4

7 Findings **5**

7.1 Medium Risk . . . 5

7.1.1 Incorrect use of `MaxTotalUsdAmount` in `makePurchase()` can leave the user unprotected against sandwich attacks . . . 5

7.1.2 High default swap slippage enables sandwich attacks . . . 5

7.1.3 If PinShop oracle is updated, the Purchase agent will not perform any more purchases as the oracle is hardcoded . . . 6

7.2 Low Risk . . . 6

7.2.1 Rewards conversion vulnerable to flashloan manipulation . . . 6

7.2.2 Potential missmatch between `tokenId` and `listingId.toknid` when setting Purchase configs can stop the contract from compounding . . . 7

7.3 Informational . . . 8

7.3.1 Unused parameter in performUpkeep function . . . 8

7.3.2 Hardcoded Sepolia addresses create deployment risks . . . 8

1 Introduction

A time-boxed review of the Stage-0 for Pinlink with focus on the security aspects of the smart contracts.

2 About Prism

Prism delivers specialized security solutions for blockchain and AI companies. We go beyond traditional audits, offering bespoke penetration testing, adversary simulation, and AI security solutions to meet the needs of every client. With tailored services and best-in-class expertise, we safeguard your business against the most sophisticated threats, allowing you to focus on innovation.

Learn more about us at prismsec.xyz

3 About Pinlink

Pinlink is the First RWA-Tokenized DePIN marketplace driving down costs for AI Developers and creating new revenue for asset owners.

4 Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time and resource-bound effort to find as many vulnerabilities as possible, but there is no guarantee that all issues will be found. This security review does not guarantee against a hack. Any modifications to the code will require a new security review.

Security researchers hold no responsibility for the findings provided in this document. A security review is not an endorsement of the underlying business or product and can never be taken as a guarantee that the protocol is bug-free. This security review is focused solely on the security aspects of the smart contracts. Gas optimizations are not the primary focus, but significant inefficiencies will also be reported.

5 Risk classification

Severity level	**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 unexpected behavior with some of the protocol's functionalities that are not so critical.

5.2 Likelihood

- High: almost certain to happen, easy to perform or highly incentivized.
- Medium: only conditionally possible or incentivized, but still relatively likely
- Low: requires multiple unlikely conditions, or little-to-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 Executive Summary

Summary

Project Name	Pinlink
Repository	FasihHaider/PinLinkShop-Agent
Commit	4c46efc9ab7d189fa8bab7a869ecaf06db18cea5
Type of Project	RWA, Auto-compounding
Audit Timeline	2025-06-10 - 2025-06-15
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	2
Gas Optimizations	0
Informational	2

6.1 Files in scope

Files in scope	nSLOC
src/PurchaseAgent.sol	141
src/interfaces/IPinlinkOracle.sol	3
src/interfaces/IPinlinkShop.sol	13
Total	**157**

6.2 Architecture review

6.2.1 System overview

The `PurchaseAgent` is a Solidity smart contract project for an automated purchasing agent that operates on top of the Pinlink DeFi marketplace. The contract automatically buys fractional assets, manages rewards, and performs automated swaps.

Core Contract: PurchaseAgent

- **Main functionality**: Automated purchasing of fractional assets from Pinlink marketplace. The contract uses the earned USDC rewards to purchase more fractional assets to auto-compound the USDC rewards and increase yield overall.
- **Inheritance**: `Ownable2Step` for access control, `IERC1155Receiver` for NFT handling
- **Key integrations**: Uniswap V2 for swaps, Pinlink Oracle for pricing, Pinlink Shop for marketplace

Key Components

- **Purchase Logic**: `purchase()` and `performUpkeep()` for automated buying
- **Reward Management**: `convertRewards()` swaps USDC rewards to PIN tokens via Uniswap
- **Asset Management**: `withdrawFromShop()` and `depositToShop()` for marketplace interactions
- **Configuration**: Owner-controlled parameters for purchase amounts and slippage

External Dependencies

- **OpenZeppelin**: ERC20, ERC1155, and Ownable2Step contracts
- **Uniswap V2**: Router for USDC $\rightarrow$ PIN token swaps
- **Pinlink Protocol**: Custom shop and oracle interfaces

6.2.2 Architecture assessment

The contract operates based on a set of configurations. At a point in time, these configurations are:

- `tokenId`: The tokenId from which rewards are claimed
- `listingId`: The listing from which more fractions are purchased when the USDC rewards balance is high enough.
- `fractionsAmount`: The amount of fractions to purchase in each `performUpkeep()` transaction.
- `maxTotalUsdAmount`: Maximum USD amount willing to spend per purchase (spending limit). This applies to both `performUpkeep()` and `makePurchase()`.
- `swapSlippage`: initialized to 5% in the variable declaration

6.2.3 User flow

- The user creates (deploys) a Purchase Agent, configured with the configs above.
- The user makes an initial purchase of fractions from the listing id
- The owner can chose to convert USDC rewards into PIN tokens that will be later used to purchase more fractions.
- Using Chainlink Automation, an agent calls periodically `performUpkeep()`. This does the following:
  - claims rewards
  - converts USDC into PIN tokens if the USDC in balance is enough to purchase at least one fraction from the `listingId`
  - If the total USDC balance in the PurchaseAgent contract is enough to purchase `fractionsAmount` of `tokenId`, then the purchase is made.

7 Findings

7.1 Medium Risk

7.1.1 Incorrect use of `MaxTotalUsdAmount` in `makePurchase()` can leave the user unprotected against sandwich attacks

Context

The makePurchase function uses the globally configured maxTotalUsdAmount with a different _fractionsAmount parameter, creating a scaling mismatch. The maxTotalUsdAmount is linked to the configured fractionsAmount, but the function allows purchasing different amounts without adjusting the maximum USD limit proportionally.

Root cause

Location: `src/PurchaseAgent.sol` in function `makePurchase()`.

`maxTotalUsdAmount` is linked to the configured fractionsAmount, but here it is used with a different `_fractionsAmount`.

```
function makePurchase(uint256 _fractionsAmount) external onlyOwner {
    uint256 maxTotalPinAmount = oracle.convertFromUsd(address(PIN), maxTotalUsdAmount);
    if (PIN.balanceOf(address(this)) <= maxTotalPinAmount) {
        revert NotEnoughTokens();
    }
    shop.purchase(listingId, _fractionsAmount, maxTotalPinAmount);
    // ...
}
```

Impact

If `_fractionsAmount > fractionsAmount`, the purchase will fail. In the opposite case, the purchase will be frontrun, as the slippage is set referenced to the `maxTotalUsdAmount`, which is linked to the configured `fractionsAmount`.

Mitigation

Either this function should also accept `maxTotalUsdAmount` as an argument, or the `maxTotalUsdAmount` should be declared "per-fraction", so `maxUsdAmountPerFraction` and the purchase should be made with `maxTotalUsdAmount = _fractionsAmount * maxUsdAmountPerFraction`. This will require updating the other usages of `maxTotalUsdAmount` in the contract.

7.1.2 High default swap slippage enables sandwich attacks

Context

The contract sets a default swap slippage of 5% for USDC to PIN token swaps. This high slippage tolerance makes users vulnerable to frontrunning attacks during reward conversion, where attackers can exploit the generous slippage to extract value.

Root cause

Location: `src/PurchaseAgent.sol` in contract declaration.

The trading swap for PIN token is currently set to 0%. However, the default slippage in this contract is set to 5% which is high. This high slippage can lead to unexpected losses for the users that are frontrun when swapping their USDC rewards to PIN.

```
contract PurchaseAgent is Ownable2Step, IERC1155Receiver {
    // ...
    uint256 swapSlippage = 500; // 5%
    // ...
}
```

Impact

The probability of being frontrun is high, but the impact is medium. Users will lose value when their USDC rewards are swapped to PIN tokens.

Mitigation

Don't set a default slippage, and require the owner to set it explicitly in the constructor.

7.1.3 If PinShop oracle is updated, the Purchase agent will not perform any more purchases as the oracle is hardcoded

Context

The PurchaseAgent contract hardcodes an oracle address that may become misaligned with the oracle used by the Pinlink shop. If the shop updates its oracle reference, this contract will continue using the old oracle, potentially leading to price discrepancies and failed transactions.

Root cause

Location: `src/PurchaseAgent.sol` in contract declaration.

The oracle address is not hardcoded in the pinlinkshop. If the oracle in pinshop changes, the purchase agent will start reading prices from a different oracle than the pinshop. In the worst case, the old oracle will not be updated anymore, so the price used by this contract would be dangerously outdated.

```
contract PurchaseAgent is Ownable2Step, IERC1155Receiver {
    // ...
    IPinlinkOracle public constant oracle = IPinlinkOracle(0x8aBfDd3B1c76682a3340ef1Ae5C8c353ec9011aA);
    ↪  // Sepolia
    // ...
}
```

Impact

If the oracle is changed in the pinlinkshop, the performUpkeep() function can revert when trying to purchase fractions, making this contract unusable.

Mitigation

Make the `oracle` not immutable, and include a function to update the oracle matching whatever new oracle is in the pinshop.

7.2 Low Risk

7.2.1 Rewards conversion vulnerable to flashloan manipulation

Context

The _convertRewards function is vulnerable to price manipulation attacks through flashloans. An attacker can manipulate the USDC/PIN exchange rate, trigger the reward conversion through performUpkeep(), and profit from the inflated slippage tolerance, causing losses to the contract users.

Root cause

Location: `src/PurchaseAgent.sol` in function `_convertRewards()`.

The lack of access control in performUpkeep(), combined with the wrong slippage checks inside this function can lead to a frontrun attack where an attacker flashloans USDC, inflates the PIN price, calls performUpkeep() to convert the rewards, and then repays the flashloan. This slippage check has no effect, as the getAmountsOut is made in the same transaction as the `performUpkeep()`, so the amountOut will be already altered by the attacker.

```
function _convertRewards() private {
    uint256 amountIn = IERC20(USDC).balanceOf(address(this));

    address[] memory path = new address[](3);
    path[0] = address(USDC);
    path[1] = WETH;
    path[2] = address(PIN);

    uint256 amountOut = IUniswapV2Router02(UniswapV2Router).getAmountsOut(amountIn, path)[1];
    uint256 minOut = (amountOut * (10000 - swapSlippage)) / 10000;

    IERC20(USDC).approve(address(UniswapV2Router), amountIn);

    // ...
    IUniswapV2Router02(UniswapV2Router).swapExactTokensForTokensSupportingFeeOnTransferTokens(
        amountIn, minOut, path, address(this), block.timestamp
    );
    // ...
}
```

Impact

There is no economic incentive for the attacker besides griefing the user, but the impact can be high in terms of value lost during reward conversion.

Mitigation

Not trivial. The minOut should be passed as an argument from outside the blockchain, to make sure it is not frontrun. Alternatively, the amount out could be estimated by the PinOracle, but this may also fail if the oracle is out of sync with the Uniswap pool.

7.2.2 Potential missmatch between `tokenId` and `listingId.toknid` when setting Purchase configs can stop the contract from compounding

Context

The setPurchaseConfig function accepts both a tokenId and listingId parameter but doesn't validate that the listingId corresponds to the provided tokenId. This can lead to configuration mismatches that break the purchase agent's functionality.

Root cause

Location: `src/PurchaseAgent.sol` in function `setPurchaseConfig()`.

The user passes a tokenId and a listingId to this function, but there is no check that the listingId corresponds to the provided tokenId.

```
function setPurchaseConfig(
    uint256 _tokenId,
    bytes32 _listingId,
    uint256 _fractionsAmount,
    uint256 _maxTotalUsdAmount
) external onlyOwner {
    tokenId = _tokenId;
    listingId = _listingId;
    fractionsAmount = _fractionsAmount;
    maxTotalUsdAmount = _maxTotalUsdAmount;
    emit PurchaseConfigUpdated(_tokenId, _listingId, _fractionsAmount, _maxTotalUsdAmount);
}
```

Impact

If mismatching arguments are provided, the purchase agent won't work, and rewards won't auto-compound.

Mitigation

Don't allow the tokenId as an argument. For a given listingId, the tokenId is always the same.

7.3 Informational

7.3.1 Unused parameter in performUpkeep function

Context

The performUpkeep function declares a `bytes calldata` parameter that is never used within the function body. This creates unnecessary function signature complexity and potential confusion for developers.

Root cause

Location: `src/PurchaseAgent.sol` in function `performUpkeep()`.

Unused `bytes calldata` parameter in the function signature.

```
function performUpkeep(bytes calldata) external {
    claimRewards();
    IPinlinkShop.Listing memory listing = shop.getListing(listingId);
    // ...
}
```

Mitigation

The input parameter can be removed to simplify the function signature.

7.3.2 Hardcoded Sepolia addresses create deployment risks

Context

The contract has hardcoded Sepolia testnet addresses for critical external dependencies including WETH, USDC, UniswapV2Router, and the Pinlink shop. This creates deployment and maintenance risks when moving between networks or when external contracts are updated.

Root cause

Location: `src/PurchaseAgent.sol` in contract declaration.

Hardcoded addresses make the contract network-specific and inflexible for deployment across different environments.

```
contract PurchaseAgent is Ownable2Step, IERC1155Receiver {
    address public constant WETH = 0xfFf9976782d46CC05630D1f6eBAb18b2324d6B14; // Sepolia
    address public constant USDC = 0x31548a5E3504BffD5CD9a350d1DFcc66c1ab7Ddb; // Sepolia
    address public constant UniswapV2Router = 0xeE567Fe1712Faf6149d80dA1E6934E354124CfE3; // Sepolia

    IPinlinkShop public constant shop = IPinlinkShop(0xF768Ab9369Debfa85957C93f651BA38358b829d5); //
    ↪  Sepolia
    // ...
}
```

Impact

Deployment risks and maintenance overhead when moving to different networks or when external contract addresses change.

Mitigation

Consider making these addresses configurable through constructor parameters or setter functions for better deployment flexibility.