Pashov Audit Group

Bio
Security Review

December 15th 2025 - December 17th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out 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. Risk Classification

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

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

4. About Bio Launchpad

Bio Launchpad is a decentralized launchpad platform that enables the creation, launch, and trading of AI agent tokens on the blockchain. Built with Foundry and Solidity, it provides a complete ecosystem for launching and managing agent-based projects with built-in vesting, staking, and liquidity mechanisms.

5. Executive Summary

A time-boxed security review of the **bio-xyz/launchpad-agent-evm** repository was done by Pashov Audit Group, during which **0xl33**, **btk**, **t.aksoy**, **Aamirusmani1552** engaged to review **Bio Launchpad**. A total of **6** issues were uncovered.

Protocol Summary

Project Name	Bio Launchpad
Protocol Type	Launchpad
Timeline	December 15th 2025 - December 17th 2025

Review commit hash:

- 9916a13782456ed3269e1489343ac80f9f736fc7 (bio-xyz/launchpad-agent-evm)

Fixes review commit hash:

- 566007971deffe6c1b26099b8cfed2ad21b430cb (bio-xyz/launchpad-agent-evm)

Scope

`Airdrop.sol` `Execute.sol`

6. Findings

Findings count

Severity	Amount
Medium	1
Low	5
Total findings	**6**

Summary of findings

ID	Title	Severity	Status
[M-01]	Cannot recover unclaimed airdrop tokens	Medium	**Resolved**
[L-01]	Arithmetic underflow when `startTime` is in the future	Low	**Resolved**
[L-02]	`mintParams` array limit inconsistent with documentation	Low	**Resolved**
[L-03]	Using `block.timestamp` for deadline makes it ineffective	Low	**Resolved**
[L-04]	`execute()` with `isFailed == true` always reverts	Low	**Resolved**
[L-05]	Recipient never receives leftover tokens	Low	**Resolved**

Medium findings

[M-01] Cannot recover unclaimed airdrop tokens

Severity

Impact: High

Likelihood: Low

Description

The Airdrop contract includes a recoverWrongToken function that allows the owner to withdraw tokens mistakenly sent to the contract, except for the main airdrop token. However, there is no mechanism for the owner to recover unclaimed airdrop tokens. Any airdrop tokens not claimed by users will remain locked in the contract indefinitely. This is likely, as some users may never claim their allocation.

Recommendations

Add a function that allows the owner to withdraw unclaimed airdrop tokens after a certain period.

Low findings

[L-01] Arithmetic underflow when `startTime` is in the future

The `Airdrop` contract allows setting `startTime` to a future timestamp during initialization:

```
// In Airdrop.initialize()
startTime = _startTime == 0 ? block.timestamp : _startTime;
```

However, when users call `claim()` or `getClaimableAmount()` before `startTime`, the elapsed time calculation causes an arithmetic underflow:

```
// In claim() and getClaimableAmount()
uint256 elapsed = block.timestamp - startTime; // underflows if block.timestamp < startTime
```

Users attempting to claim or check claimable amounts before `startTime` receive a confusing panic error instead of a clear error message indicating the airdrop hasn't started yet.

Recommendation: Add a check at the beginning of both functions.

```
if (block.timestamp < startTime) revert AirdropNotStarted();
```

For `getClaimableAmount()`, alternatively return `0` to indicate nothing is claimable yet.

[L-02] `mintParams` array limit inconsistent with documentation

The `Execute.execute()` function's NatSpec states that `mintParams` can have a maximum of 10 elements:

```
/// @param mintParams Array of parameters for minting LP positions (max 10)
```

However, the implementation enforces a maximum of 9 elements:

```
require(mintParams.length < 10, "above max length");
```

This inconsistency between documentation and implementation means admins cannot use the full capacity of 10 LP positions as documented.

Recommendation: Change the validation to allow up to 10 elements: `require(mintParams.length <= 10, "above max length");`.

[L-03] Using `block.timestamp` for deadline makes it ineffective

In `Execute._mintLP()`, the deadline for minting LP positions is set using `block.timestamp`:

```
    $.nfpManager.mint({
        params: INfpm.MintParams({
            token0: token0,
            token1: token1,
            tickSpacing: mintParams[i].tickSpacing,
            tickLower: tickLower,
            tickUpper: tickUpper,
            amount0Desired: amount0,
            amount1Desired: amount1,
            amount0Min: 0,
            amount1Min: 0,
            recipient: mintParams[i].recipient,
@>          deadline: block.timestamp + 5 minutes,
            sqrtPriceX96: 0
        })
    });
```

Since `block.timestamp` is evaluated at execution time, the deadline will always be 5 minutes in the future. This defeats the purpose of the deadline parameter, as validators can hold the transaction in the mempool indefinitely, and it will always pass the deadline check when executed.

If the transaction sits in the mempool for an extended period, the admin may face gas price volatility and execution at a much later time than intended. While the risk is limited due to the pool being created in the same transaction (preventing MEV), this still represents an operational issue.

Recommendation: Add a `deadline` parameter to the `execute()` function that the admin can set, and pass it through to the `mint()` call instead of using `block.timestamp`.

[L-04] `execute()` with `isFailed == true` always reverts

The `Execute.execute()` function accepts `isFailed` parameter but will always revert when called with `isFailed == true`, due to agent not being created:

```
// in Launch._onLaunchSuccess()
@>      if (isFirstLaunch && !_isFailed) {
            // ... other logic ...

@>          agentToken = IAgentFactory($.agentFactoryAddress).executeFactoryApplicationSalt(id,
mintData, salt);

            require(agentToken != address(0), "Agent token creation failed");

            // Store the created agent token address
@>          $.agentTokenAddress = agentToken;
            $.vestingStartTime = block.timestamp;
        }
```

```
// in execute(), after onLaunchSuccess() call
        address agent = launch.agentTokenAddress(); // returns address(0)
```

Since no agent token is created when `isFailed == true`, the `agent` address is `address(0)` and the transaction will always revert. The place where it reverts depends on which actions the admin wants to execute, but even when admin calls the function with the bare minimum parameters, the function will still revert, as proven by the test below.

Proof of concept:

1. Add the below test to `Execute.t.sol`.
2. Run the test with the following command: `forge test --mt testExecuteWithIsFailedTrue -vvvvvv`.
3. You will see that the test reverts in this line (in `execute()`): `leftover = IERC20(agent).balanceOf({account: address(this)});`

```
    function testExecuteWithIsFailedTrue() public {
        vm.warp(block.timestamp + 8 days);

        ISwapRouter.ExactInputSingleParams memory params = ISwapRouter.ExactInputSingleParams({
            tokenIn: address(1),
            tokenOut: address(2),
            tickSpacing: 200,
            recipient: multiSigAddress,
            deadline: block.timestamp,
            amountIn: 0,
            amountOutMinimum: 0,
            sqrtPriceLimitX96: 0
        });

        ICLPoolLauncher.MintParams[] memory mintP = new ICLPoolLauncher.MintParams[](0);
        Execute.Lock[] memory lock = new Execute.Lock[](0);
        Execute.Vesting[] memory vest = new Execute.Vesting[](0);

        deal(address(usdc), address(launch), 50_000 * 1e6);

        Execute.AirdropParams memory airdropParams = Execute.AirdropParams({
            merkleRoot: bytes32(0),
            receiver: address(0),
            lockUpDuration: 0,
            totalAllocation: 0,
            owner: address(0),
            startTime: 0
        });

        vm.startPrank(multiSigAddress);

        uint256 launchId = Launch(launch).launchId();

        vm.expectRevert();
        execute.execute(
            true,
            launchId,
            200,
            0,
            50_000 * 1e6,
            address(33),
            lock,
            bytes32(0),
            0,
            vest,
            0,
            mintP,
            params,
            airdropParams
        );

        vm.stopPrank();
    }
```

Recommendation: Either remove the `isFailed` parameter entirely and hardcode it to `false`, or add an early return after the `onLaunchSuccess()` call when `isFailed == true`.

[L-05] Recipient never receives leftover tokens

The `Execute.sol` contract includes a state variable `recipient` intended to hold the address of a designated wallet (e.g., a multisig) for receiving leftover funds. This address is configured during `initialize` and can be updated via the `updateRecipient` function.

However, the actual logic for handling leftover tokens at the end of the `execute` function ignores this configured `recipient`. Instead, it transfers all remaining `baseToken` and `agent` tokens directly to `msg.sender`.

```
// At the end of the execute function
uint256 leftover = IERC20(baseToken).balanceOf({account: address(this)});
if (leftover > 0) {
    // BUG: "to" should be the stored recipient, not msg.sender
    IERC20(baseToken).safeTransfer({to: msg.sender, value: leftover});
}

leftover = IERC20(agent).balanceOf({account: address(this)});
if (leftover > 0) {
    // BUG: "to" should be the stored recipient, not msg.sender
    IERC20(agent).safeTransfer({to: msg.sender, value: leftover});
}
```

Recommendation

Modify the token transfer calls to use the `recipient` address from storage instead of `msg.sender`.

```
- IERC20(baseToken).safeTransfer({to: msg.sender, value: leftover});
+ IERC20(baseToken).safeTransfer({to: _getExecuteStorage().recipient, value: leftover});

- IERC20(agent).safeTransfer({to: msg.sender, value: leftover});
+ IERC20(agent).safeTransfer({to: _getExecuteStorage().recipient, value: leftover});
```