Goose Run Security Audit

Report Version 1.0

October 31, 2024

Conducted by **Hunter Security**

Table of Contents

1 About Hunter Security

Hunter Security is an industry-leading smart contract security auditing firm. Having conducted over 100 security audits protecting over $1B of TVL, our team always strives to deliver top-quality security services to the best DeFi protocols. For security audit inquiries, you can reach out on Telegram or Twitter at *@georgehntr*.

2 Disclaimer

Audits are a time-, resource-, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

3 Risk classification

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

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Hunter Security was engaged by Goose Run to review a part of the Fair Launch smart contract protocol.

Overview

Project Name	Goose Run
Repository	https://github.com/fairlaunchrc/fairlaunch_contracts
Commit hash	cb3b10370a029e39694dde45c51027bfb5b1cf75
Resolution	5c6d82a460035735301f7531b842a58dfe68188c
Methods	Manual review & testing

Scope

contracts/src/RaffleVault.sol
contracts/src/Swapper.sol
contracts/src/VotingDistributor.sol

Issues Found

High risk	0
Medium risk	0
Low risk	4
Informational	10

5 Findings

5.1 Low

5.1.1 Users will lose lootboxes if Entropy address is changed while pending requests

Severity: Low

Context: Swapper.sol#L115

Description: When a user wants to open their lootboxes, they need to first request randomness via the *RaffleVault.openLootBoxes* method which would call the Entropy smart contract address set in the Swapper contract. Then the Entropy will invoke the specified callback function on the RaffleVault passing the generated random number.

The problem is that the owner can change the address of the Entropy while there's a pending request for randomness. This would lead to the user who has created the request losing their lootboxes since the old Entropy contract will not be able to call the callback function.

Recommendation: Consider the likelihood and impact of such scenario happening. A possible solution would be to implement a counter of the current pending requests and only allow a change of the Entropy contract if there are no pending requests in any raffle vault (which may not be practical or worth the additional logic).

Resolution: Acknowledged.

5.1.2 Sweep may return a value different from the one obtained from the swap

Severity: Low

Context: Swapper.sol#L327

Description: In *Swapper.sellToken*, a user's specified amount of launch tokens is exchanged for its corresponding quote token. The amount of quote tokens obtained in that trade is considered to be the balance of the Swapper contract as it is the address passed as recipient to *pool.swap*.

A potential scenario is that except the quote tokens received from the pool from that swap, there are other already present quote tokens in the contract's balance. This could happen if the caller wants to execute a multicall for a sell and buy (where the launch token is the same), and has transferred the necessary amount of launch tokens for the buy operation to the Swapper upfront (i.e. before executing the sell).

In that case, *Swapper.sellToken* would return the retrieved amount of launch tokens from the swap **plus** the amount of launch tokens the user has sent upfront for the following buy operation in the multicall. The result would be that the multicall will fail as the buy would not be executed due to lack of funds (the sent amount of tokens upfront would be returned to the user at the end of the sell).

Recommendation: Consider setting *amountOut* to the return value of *_extractFee* in *sellToken* as is done in *buyToken*.

Resolution: Acknowledged.

5.1.3 Selling launch tokens for WETH won't work for contracts with no receive function

Severity: Low

Context: Swapper.sol#L375

Description: When selling a launch token via *Swapper.sellToken*, if the quote token is WETH it would be unwrapped and sent as native tokens rather than as an ERC20.

This could be a problem if a smart contract caller that does not implement a *receive* function wants to use this feature as the WETH will be unwrapped regardless of the user's preference.

Recommendation: Consider always sending the swept amount in the form of an ERC20 token (i.e. WETH) and let the user decide whether they would like to unwrap WETH for native tokens.

Resolution: Acknowledged.

5.1.4 Unsafe approve when sending quote token fee to the voting distributor

Severity: Low

Context: Swapper.sol#L423

Description: The fees collected on each swap are sent to their recipient by minting corresponding shares in the FeeVault contract and letting them claim from there. However, the fee going to the quote token's VotingDistribution contract is sent directly to it by first approving the amount of tokens and then calling the *addVoteIncentiveToCurrentEpoch* method.

The problem is that the *approve* function is called on the quote token which may not be a standard ERC20 as some tokens such as USDT do not return a boolean or implement the following check:

```
require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
```

Recommendation: Consider using OpenZeppelin's SafeERC20 library and its *forceApprove* method which solves both of the aforementioned issues.

Resolution: Resolved.

5.2 Informational

5.2.1 Typographical mistakes, non-critical issues and code-style suggestions

Severity: Informational

Context: .

Description: The contracts contain one or more typographical mistakes, non-critical issues and code-style suggestions. In an effort to keep the report size reasonable, we enumerate these below:

1. Typographical mistake - the symbol of the buy tracker should be "GSBFT" instead of "GSSFT".
2. Typographical mistake on line 37 in Swapper.sol - "is" should be "if".
3. The *dataForSequenceNumber* mapping may have the entropy contract address as first key since changing the entropy in the Swapper will lead to using other sequence numbers, eventually overriding already used ones.

4. Unsafe casts to *int256* in *endDayTimestampAtOffset*.
5. Consider the gas limits for the callback function on each blockchain as advised in the oracle's documentation - https://docs.pyth.network/entropy/best-practices#limit-gas-usage-on-the-callback.
6. The first *if* check in the *_pay* function is redundant since WETH is an ERC20 token as well.
7. Insufficient input validation for *onlyOwner* setter methods - consider enforcing strict value ranges (i.e. protocol fee recipient cannot be the 0 address, order of thresholds, etc.).
8. Missing 2-step ownership transfer pattern.
9. *prizeAmount.toUint160()* may overflow in rare cases such as high-number token decimals.
10. Fee transfers may revert if the amount is 0 for some ERC20 tokens.

Recommendation: Consider fixing the above typographical mistakes, non-critical issues and code-style suggestions.

Resolution: Resolved.