Preduck Games (EVM)

Security Audit Report

December 8, 2025

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

BugBlow utilized a widely adopted approach to performing a security audit. Below is a breakout of how our team was able to identify and exploit vulnerabilities.

1.2.1 Code and architecture review:

- ➢ Review the source code.
- ➢ Read the project's documentation.
- ➢ Review the project’s architecture.

Stage goals

- ➢ Build a good understanding of how the application works.

1.2.2 Check code against vulnerability checklist:

- ➢ Understand the project’s critical assets, resources, and security requirements.
- ➢ Manual check for vulnerabilities based on the auditor's checklist.
- ➢ Run static analyzers.

Stage goals

- Identify and eliminate typical vulnerabilities (gas limit, replay, flash-loans attack, etc.)

1.2.3 Threat & Attack Simulation:

- Analyze the project's critical assets, resources and security requirements.
- Exploit the found weaknesses in a safe local environment.
- Document the performed work.

Stage goals

- Identify vulnerabilities that are not listed in static analyzers that would likely be exploited by hackers.
- Develop Proof of Concept (PoC).

1.2.4 Report found vulnerabilities:

- Discuss the found issues with developers
- Show remediations.
- Write an initial audit report.

Stage goals

- Verify that all found issues are relevant.

1.2.5 Fix bugs & re-audit code:

- Help developers fix bugs.
- Re-audit the updated code again.

Stage goals

- Double-check that the found vulnerabilities or bugs were fixed and were fixed correctly.

1.2.4. Deliver final report:

- The Client deploys the re-audited version of the code
- The final report is issued for the Client.

Stage goals

- Provide the Client with the final report that serves as security proof.

Severity classification

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Vulnerabilities leading to assets theft, fund access locking, or any other loss of funds.
High	Vulnerabilities that can trigger contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Vulnerabilities that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Vulnerabilities that do not have a significant immediate impact and could be easily fixed.

1.3 Project Overview

Preduck Games is a a decentralized prediction market built on BNB Chain, where users place bets on whether the price of an asset (typically BNB/USD from Chainlink) will rise or fall within a fixed time interval.

The Preduck Games implementation extends the original PancakeSwap prediction (V2) market architecture while introducing additional functionality tailored for backend-assisted operations and enhanced ecosystem integration. Similar to the original protocol, the system enables users to place bull/bear bets on future price movements based on Chainlink oracle data, with automated round execution and proportional reward distribution.

Key Differences from the Original Pancake Prediction V2

1. Backend-Signature Claiming (Referral / External Rewards)
   A new EIP-712–based mechanism allows users to claim referral or external bonus rewards through signatures generated by a trusted backend (backendSigner).
   This functionality does *not* exist in the original PancakeSwap contract.

2. Added backendSigner Role & Additional State Tracking
   The modified version introduces the backendSigner address and maintains user-specific nonces to prevent replay attacks during signature-based claims.
   These fields and logic are absent in the original.

3. Extended Treasury Logic
   The custom contract allows backend-validated payouts from the treasury (claimReferralReward), whereas the original Pancake contract only supports a single admin-controlled treasury withdrawal (claimTreasury).

4. EIP-712 Domain Support
   The custom version inherits EIP712, making it suitable for off-chain signed rewards and integrations with external systems — functionality entirely missing in the original.

5. Minor Structural Adjustments in Round Lifecycle Management
   Core mechanics — start, lock, end, reward calculation — largely mirror the original.
   However, the modified version includes additional events, extended return structures, and batch-style queries for analytics.

1.4 Project Dashboard

Project Summary

Title	Preduck Games Security Audit
Client name	Preduck Games
Project name	Preduck Games Smart Contracts (EVM) https://preduck.games

Timeline	20.11.2025 - 03.12.2025
Number of Auditors	3

Project Last Log

Date	Commit Hash	Note
10.06.2025	d022c3c78a26be01e7113b4a87c10e81c7b7a5a6	fix: remove old

Project Scope

The audit covered the following smart contract files.

File name
src/PancakePredictionV2.sol
src/utils/TimeSeriesViewer.sol

1.5 Summary of findings

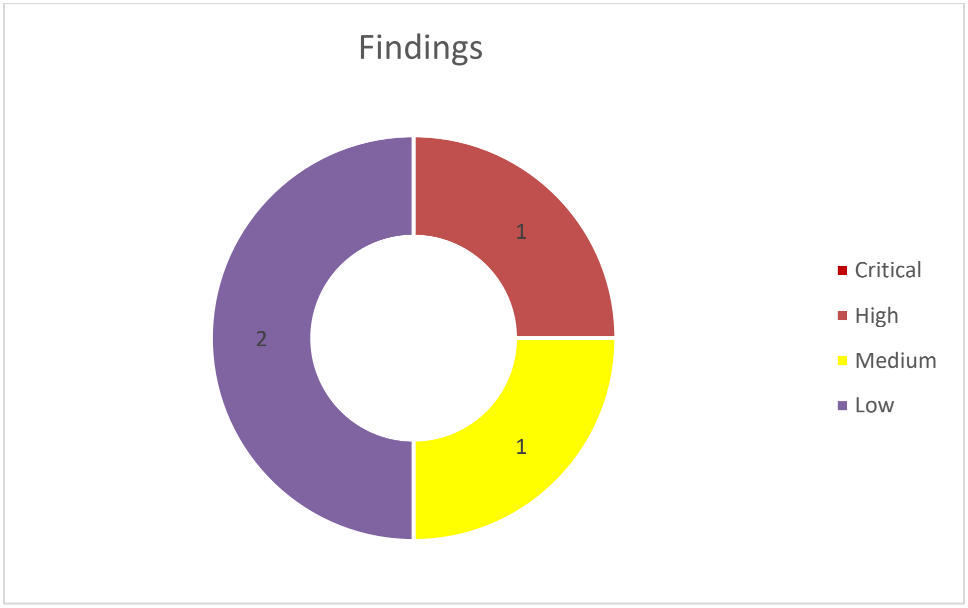

Figure 1. Findings chart

Severity	# of Findings
CRITICAL	0
HIGH	1
MEDIUM	1
LOW	2

2. FINDINGS REPORT

2.1 Critical

2.2 High

2.2.1 Oracle Freshness Check Does Not Work (Stale Price Accepted)

Severity: High

Status: Fixed

Location: _getPriceFromOracle()

Description

The freshness check for the oracle price contains a critical logical flaw that causes the contract to accept stale oracle data — including prices from minutes, hours, or even days ago.
As a result, the protocol may settle rounds using outdated prices, leading to incorrect payouts and direct financial loss for users.
This also opens an attack vector commonly known as a stale update / oracle freeze exploit, which is severe because the oracle price is the single source of truth for the prediction market.

Chainlink guarantees the *accuracy* of data, but Chainlink explicitly does not guarantee *freshness* of timestamps.

Technical Analysis

Problematic code:

```
uint256 leastAllowedTimestamp = block.timestamp + oracleUpdateAllowance;
require(timestamp <= leastAllowedTimestamp, "Oracle update exceeded max timestamp allowance");
```

This logic is inverted.
$block.timestamp + allowance >= timestamp$ is always true unless the aggregator submits a timestamp in the future.

The correct check must ensure that:

$$(block.timestamp - timestamp) <= oracleUpdateAllowance$$

Because:

- $block.timestamp$ = current block time
- $timestamp$ = last time the oracle updated
    - may be 10 seconds old
    - 50 seconds old
    - yesterday

Impact

- Rounds are settled using stale prices
- Users receive incorrect payouts
- Predictable stale-price windows allow for profitable manipulation
- Protocol trust assumptions break entirely

Fix

Replace the incorrect freshness check with a proper staleness validation:

```diff
- uint256 leastAllowedTimestamp = block.timestamp + oracleUpdateAllowance;
- require(timestamp <= leastAllowedTimestamp, "Oracle update exceeded max timestamp allowance")
+ require(block.timestamp - timestamp <= oracleUpdateAllowance, "Oracle price is stale");
```

2.3 Medium

2.3.1 Strict RoundID Check May Cause DoS

Severity: High

Status: Fixed

Location: _getPriceFromOracle()

Description

The contract enforces a strict requirement that the Chainlink roundId must always increase:

$$require(uint256(roundId) > oracleLatestRoundId, \ldots);$$

However, Chainlink does NOT guarantee that $roundId$ always increases [1]:

- roundId may stay the same for some time.
- Some feeds update sparsely or aggregate several on-chain pushes
- Certain feeds update internal aggregator phases, which may reuse roundId patterns

This creates a scenario where:

- roundId remains the same
- The contract rejects the update

- And **freezes** the prediction market

Technical Analysis

Impact

Round DoS (Denial of Service):

- executeRound() stops working
- Rounds never close
- Rounds never lock
- Rewards are never calculated
- Users cannot claim
- Contract becomes permanently stuck after the first occurrence of equal roundId

This is a **failure mode**.

Fix

Use a non-strict check:

```
- require(
-     uint256(roundId) > oracleLatestRoundId,
-     "Oracle update roundId must be larger than oracleLatestRoundId "
-);

+ require(
+     uint256(roundId) >= oracleLatestRoundId,
+     "RoundId must not decrease"
+);
```

2.4 Low

2.4.1 TimeSeriesViewer Array Index Out-of-Bounds

Severity: Low

Status: Fixed

Location: TimeSeriesViewer.viewHistoricalPrices()

Description

The function uses this cycle

```
for (uint80 i = firstRoundId; i <= lastRoundId; i++) {
    (roundIds[i], prices[i], , timestamps[i], ) = AggregatorV3Interface(aggregator).getRoundData(i);
}
```

The roundIds array is of size:

```
numberRounds = lastRoundId - firstRoundId + 1;
roundIds = new uint80[](numberRounds); // indexes: 0 .. numberRounds-1
```

But indexing $[i]$ is performed through $[firstRoundId\ \dots\ lastRoundId]$:

```
roundIds[i] = ...
```

This means:

Valid indices: $[0\ \dots\ numberRounds - 1]$

Accessed indices: $[firstRoundId\ \dots\ lastRoundId]$

For typical parameters (firstRoundId doesn't start with 0):

$firstRoundId = 100$

$lastRoundId = 110$

$numberRounds = 11$

$valid\ indexes = [0..10]$

but code uses indexes [100..110] → ALWAYS out of bounds

Impact

- The function always reverts unless firstRoundId == 0
- Any frontend or analytics tool relying on this contract cannot retrieve historical data
- Makes the contract unusable for its intended purpose

Fix

Fix the index mapping:

```
- for (uint80 i = firstRoundId; i <= lastRoundId; i++) {
-     (roundIds[i], prices[i], , timestamps[i], ) =
-         AggregatorV3Interface(aggregator).getRoundData(i);
- }

+ for (uint80 i = firstRoundId; i <= lastRoundId; i++) {
+     uint256 idx = i - firstRoundId;
+     (roundIds[idx], prices[idx], , timestamps[idx], ) =
+         AggregatorV3Interface(aggregator).getRoundData(i);
+ }
```

2.4.2 Incorrect View Treasury Share Calculation

Severity: Low

Status: Fixed

Location: getUserRewardAndTreasuryShareBatch()

Description

The The getUserRewardAndTreasuryShareBatch function is intended to compute how much of a round's treasury contribution is attributable to a specific user (treasuryShare).
However, the logic does not match the actual economic model of the protocol.

The function assumes the treasury amount is always:

$$totalTreasury \; = \; (round.totalAmount \; * \; treasuryFee) \; / \; 10000;$$

$$treasuryShare \; = \; (betInfo.amount \; * \; totalTreasury) \; / \; round.totalAmount;$$

But this assumption breaks in the following case:

House Wins scenario (no winners)

In the contract's actual logic:

$$if \; (rewardBaseCalAmount \; == \; 0) \; \{$$

$$treasuryAmt \; = \; round.totalAmount; \; // \; 100\% \; to \; treasury$$

$$\}$$

Meaning:

- All funds go to treasury, not just the fee.
- But the view function still assumes treasury = fee only.
- Therefore, the reported treasuryShare is incorrect.

Impact

This results in misleading data:

- UI and API show incorrect values.
- Any analytics, dashboards, bots, or backend logic relying on this function produce incorrect output.
- External systems cannot accurately display the protocol's treasury distribution.
- Users may perceive incorrect results during loss/refund scenarios.

While this is not a fund-loss vulnerability, it is a significant correctness issue affecting external integrations and user trust.

Fix

Before calculating treasureShare, the read size of totalTreasure needs to be calculated:

```
uint256 totalTreasury = (
round.rewardBaseCalAmount == 0
? round.totalAmount // House wins: all amount goes to treasury.
: (round.totalAmount * treasuryFee) / 10000
);
```

And then calculate treasureShare:

```
results[i].treasuryShare =
(betInfo.amount * totalTreasury) / round.totalAmount;
```

CONCLUSION

The reviewed contracts contain several high-priority correctness and oracle-integration flaws that directly affect protocol safety and liveness. Incorrect handling of Chainlink price freshness and roundId logic can lead to stale-price settlements and even a protocol DoS. Additionally, indexing errors in the TimeSeriesViewer contract prevent reliable historical data retrieval, indicating gaps in testing and validation of auxiliary components. Fixing these issues will significantly increase the protocol's reliability and security.

REFERENCES

[1] https://docs.chain.link/data-feeds/historical-data