Comprehensive Security Audit Report

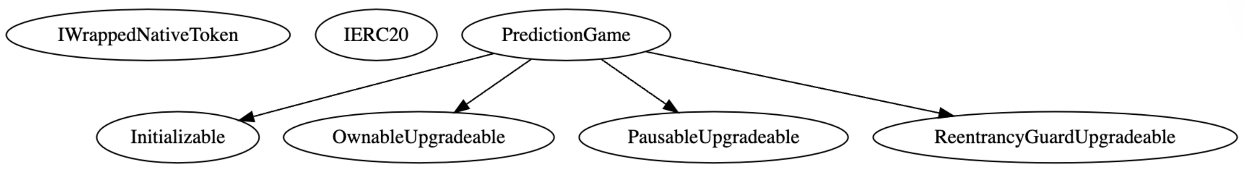
Project Name: Up Down
Audit Date: October 17, 2025
Auditor: Senior Solidity Security Auditor
Scope: PredictionGame.sol, MockERC20.sol, WrappedNativeToken.sol(Solidity ^0.8.28)
Codebase: Provided by file
Methodology: Line-by-line manual review, automated analysis, architectural assessment

EXECUTIVE SUMMARY

Risk Summary
• Total Contracts Analyzed: 3 (PredictionGame.sol, WrappedNativeToken.sol, MockERC20.sol)
• High Vulnerabilities: 0
• Medium Vulnerabilities: 2
• Low/Informational: 5

Deployment Readiness
NOT READY FOR MAINNET - Vulnerabilities must be resolved first.

CONTRACT INHERITANCE TREE



OWNERSHIP PRIVILEGES

PredictionGame.sol
OWNER_ROLE: 1. The owner can pause and unpause the contract.
2. The owner can modify the treasury fee up to a maximum of 10%
3. The owner can change the operator's address at any time without restrictions.
4. The owner can set interval seconds for round duration, but only when the contract is paused.
5. The owner can claim accumulated treasury funds for any payment token at any time.
6. The owner can authorize contract upgrades through the proxy pattern.

OPERATOR_ROLE: 1. The operator can initialize genesis rounds for new token pairs with initial price data.
2. The operator can execute rounds, which lock the current round, end the previous round, calculate rewards, and start a new round.
3. The operator can pause the contract, but cannot unpause it (only the owner can unpause).
4. The operator has exclusive control over all price feeds and round progression, creating a single point of failure for price data integrity.

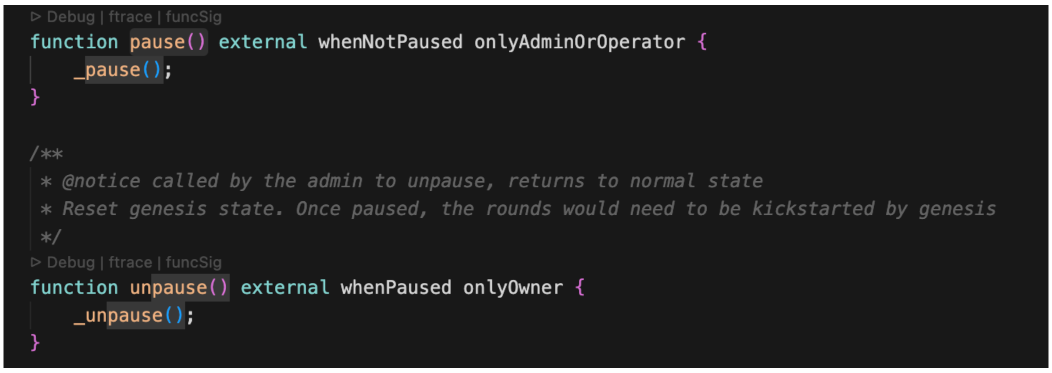
UPGRADER_ROLE (Proxy Admin): 1. The proxy admin can upgrade the contract implementation to completely new logic while preserving storage state.
2. The proxy admin can replace all contract behavior without user consent or time delays.

DETAILED VULNERABILITY ANALYSIS

1- Title: Missing Cooldown on Pause/Unpause Enables Rapid Toggle Attacks

Severity: MEDIUM

File and Code Location: PredictionGame.sol/Pause() and Unpause(), Lines 639-647



```
▷ Debug | ftrace | funcSig
function pause() external whenNotPaused onlyAdminOrOperator {
    _pause();
}

/**
 * @notice called by the admin to unpause, returns to normal state
 * Reset genesis state. Once paused, the rounds would need to be kickstarted by genesis
 */
▷ Debug | ftrace | funcSig
function unpause() external whenPaused onlyOwner {
    _unpause();
}
```

Description: Pause and unpause functions lack time-based restrictions, enabling unlimited state toggling without cooldown periods. This allows operators to repeatedly pause operations or selectively block transactions by pausing before execution. The absence of rate limiting creates potential for operational disruption and transaction manipulation without adequate controls.

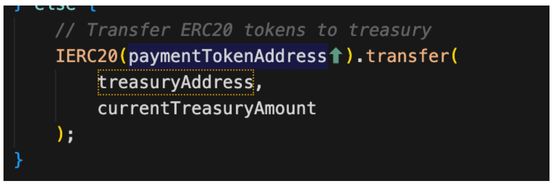
Mitigation: Implement minimum time delay between pause state changes (e.g., 1 hour cooldown). Add daily toggle limit (e.g., maximum 3 pauses per day). Consider requiring timelock or multisig for pause operations to prevent instant manipulation.

Status: OPEN

2- Title: Unchecked ERC20 Transfer Return Value in Treasury Claim Function

Severity: MEDIUM

File and Code Location: PredictionGame.sol/claimTreasury(), Lines 719, 729-732



```
} else {
    // Transfer ERC20 tokens to treasury
    IERC20(paymentTokenAddress).transfer(
        treasuryAddress,
        currentTreasuryAmount
    );
}
```

Description: The Treasury balance is cleared before ERC20 transfer, and the transfer return value is not checked. Non-standard tokens that return false instead of reverting cause silent transfer failure while the treasury balance remains zero, resulting in permanent loss of protocol revenue with no recovery mechanism available.

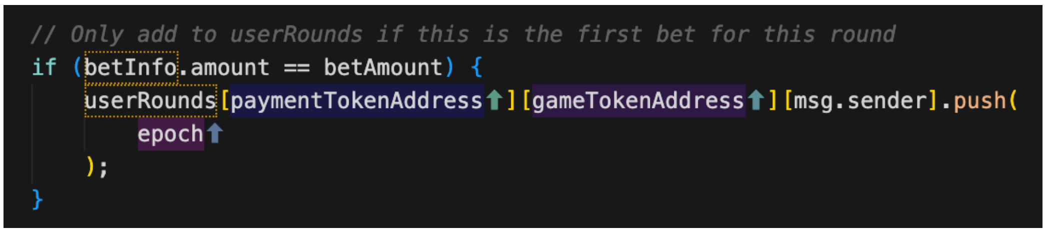
Mitigation: Use OpenZeppelin SafeERC20.safeTransfer() or add require() to check the return value. Move treasuryAmount clearing after successful transfer to follow the CEI pattern. This prevents silent failures with non-standard ERC20 tokens.

Status: OPEN

3- Title: Duplicate User Rounds Entry Due to Zero-Value Bet Logic Flaw

Severity: LOW

File and Code Location: PredictionGame.sol/betBull, Line 291-294



```
// Only add to userRounds if this is the first bet for this round
if (betInfo.amount == betAmount) {
    userRounds[paymentTokenAddress][gameTokenAddress][msg.sender].push(
        epoch
    );
}
```

Description: When minBetAmount is zero or uninitialized, users can place a zero-value bet followed by a positive bet in the same round. The condition betInfo.amount == betAmount evaluates true for both transactions, causing the epoch to be added twice to the userRounds array, creating duplicate entries and breaking user history tracking.

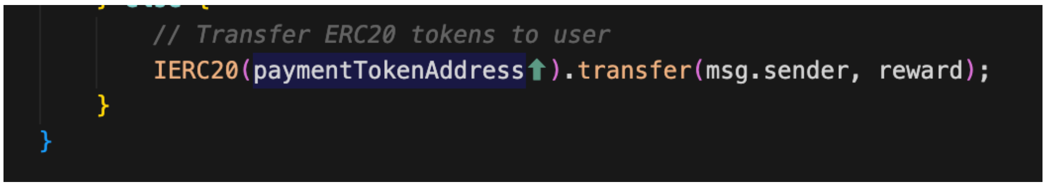
Mitigation: Add explicit zero-amount check: if (betInfo.amount == betAmount && betAmount > 0). Alternatively, enforce minBetAmount > 0 in setMinBetAmount() initialization and validation to prevent zero-value bets entirely.

Status: OPEN

4- Title: Missing Return Value Check on ERC20 Transfer in Claim Function

Severity: LOW

File and Code Location: PredictionGame.sol/claim(), Line 477



```
            // Transfer ERC20 tokens to user
            IERC20(paymentTokenAddress).transfer(msg.sender, reward);
        }
    }
```

Description: The ERC20 transfer() call does not check the return value. While most standard tokens revert on failure, some non-standard ERC20 implementations (like certain USDT versions) return false instead of reverting. If transfer fails silently, users are marked as claimed but receive no tokens, resulting in permanent fund loss.

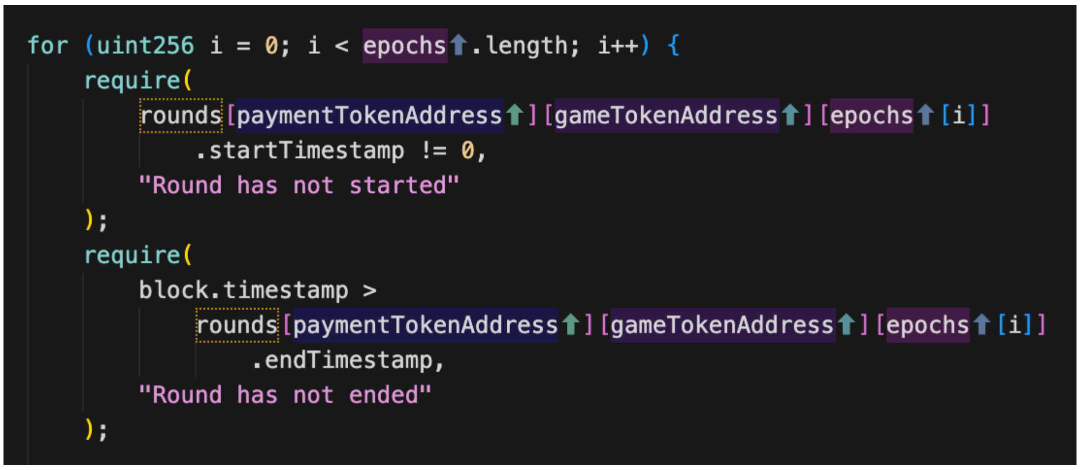
Mitigation: Use OpenZeppelin's SafeERC20 library: using SafeERC20 for IERC20; IERC20(paymentTokenAddress).safeTransfer(msg.sender, reward);. Alternatively, add explicit check: require(IERC20(...).transfer(...), "Transfer failed"); to ensure transaction reverts on failure.

Status: OPEN

5- Title: Unbounded Array Length Allows Gas Griefing in Claim Function

Severity: LOW

File and Code Location: PredictionGame.sol/claim, line 403



```
for (uint256 i = 0; i < epochs.length; i++) {
    require(
        rounds[paymentTokenAddress][gameTokenAddress][epochs[i]]
            .startTimestamp != 0,
        "Round has not started"
    );
    require(
        block.timestamp >
            rounds[paymentTokenAddress][gameTokenAddress][epochs[i]]
                .endTimestamp,
        "Round has not ended"
    );
```

Description: The claim function accepts an arbitrary-length epochs array without validation. Users or attackers can submit extremely large arrays, causing transactions to exceed block gas limits, resulting in failed claims and wasted gas fees. While funds remain safe, legitimate users may be unable to claim rewards efficiently.

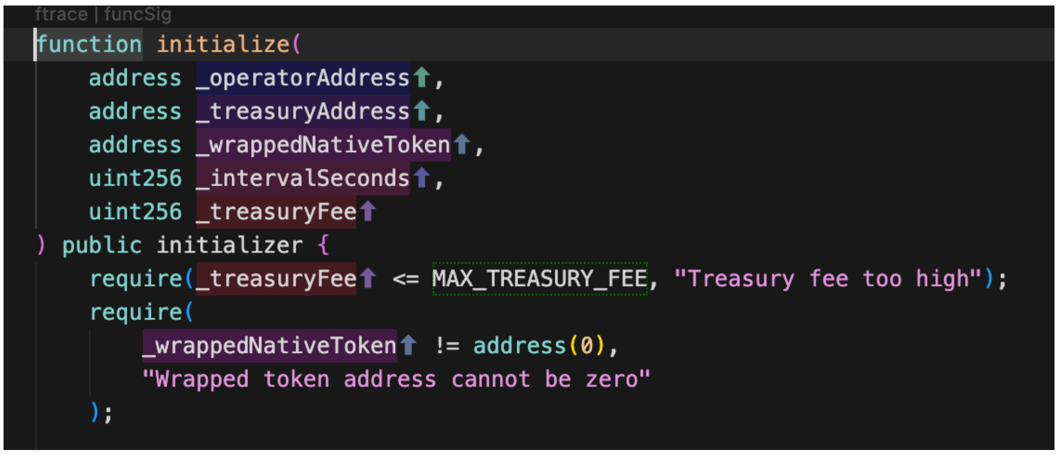
Mitigation: Add array length validation before the loop: require(epochs.length > 0 && epochs.length <= 50, "Invalid epochs length");. Choose an appropriate maximum based on gas profiling. Consider implementing pagination or batch claiming with reasonable limits.

Status: OPEN

6- Title: Missing Treasury Address Validation in Initialize Function

Severity: LOW

Code Location: PredictionGame.sol/initialize, Line 216



```
function initialize(
    address _operatorAddress,
    address _treasuryAddress,
    address _wrappedNativeToken,
    uint256 _intervalSeconds,
    uint256 _treasuryFee
) public initializer {
    require(_treasuryFee <= MAX_TREASURY_FEE, "Treasury fee too high");
    require(
        _wrappedNativeToken != address(0),
        "Wrapped token address cannot be zero"
    );
```

Description: The initialize function accepts the treasuryAddress parameter without validating it is not address(0). If the contract is initialized with zero address as the treasury, all protocol fees will be permanently sent to the burn address. No setTreasuryAddress function exists to fix this post-deployment, making it irreversible.

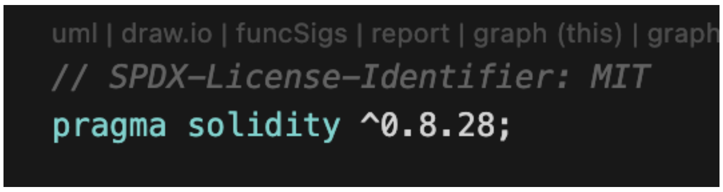
Mitigation: Add validation before assignment: require(_treasuryAddress != address(0), "Invalid treasury address"); at line 216. This ensures the treasury address is valid during initialization, preventing permanent loss of all protocol revenue.

Status: OPEN

7- Title: Floating Pragma Version

Severity: LOW

Code Location: All Contracts



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;
```

Description: All contracts use a floating pragma ^0.8.26 instead of locking to a specific compiler version. The contracts use pragma solidity ^0.8.26, which allows compilation with any version from 0.8.30 up to (but not including) 0.9.0. This creates deployment inconsistencies where different compiler versions may produce different bytecode, introducing subtle bugs or behavioral changes across deployments.

Mitigation: Lock the pragma to a specific compiler version using pragma solidity 0.8.26; to ensure consistent compilation and deployment behavior across all environments.

Status: OPEN

DISCLAIMER

This audit report is not the final deployed version of the contract. So, any changes made after the audit will be considered out of the scope of the audit, and we will not be responsible for any changes related to that.