33AUDITS & CO.

FortuneFi Audit Report

Security Audit Report

Introduction

A security review of the FortuneFi Perpetual Raffle was done, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits & Co with Lee Faria, Radoslav Radev, and Samuel Troy Dominguez as Security Researchers.

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 vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About 33 Audits & Company

33Audits is an independent smart contract security researcher company and development group. We conduct audits as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @33audits.

About FortuneFi Perpetual Raffle

FortuneFi Perpetual Raffle is a decentralized protocol featuring pool-based architecture, fee management, swap functionality, and a vault system for its perpetual raffle and prize mechanics.

Repository: BuildTheTech/FortuneFi-Contracts

Commit: 92782a13

Scope: BuildTheTech/FortuneFi-Contracts/*

Severity Definitions

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

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Findings Summary

ID	Title	Severity	Status
[H-01]	Complete DOS of weekly draw if Jackpot winner cannot receive ETH	High	Fixed
[H-02]	`BondsAdapter.sol#completeLiquidation()` requires `msg.value` (cannot pay from existing contract balance)	High	Fixed
[H-03]	Permanent paid-purchase DoS in `EntryRouter.sol` smart contract if stored referrer cannot receive ETH	High	Fixed
[H-04]	Ticket price USD decimals are wrong in EntryRouter smart contract -> tickets cost `~$0.002` instead of `$2` due to wrong USD decimals	High	Fixed
[H-05]	Price feed decimals ignored -> mispriced ticket cost if feed decimals ≠ 8 in `EntryRouter.sol` smart contract	High	Fixed
[H-06]	Weekly VRF callback in `DrawCoordinator.sol` smart contract can permanently brick the draw if it reverts (gas / external calls)	High	Fixed
[M-01]	Inconsistent Fee on Transfer Token Handling	Medium	Fixed
[M-02]	Whale volume counts free-credit tickets as paid volume in `EntryRouter.sol` smart contract	Medium	Fixed
[M-03]	Whale status can be reached with free credits in `EntryRouter.sol` smart contract (not paid volume)	Medium	Fixed
[M-04]	Daily/Monthly draw winner can be biased (timestamp-based randomness + public trigger) in `DrawCoordinator.sol` smart contract	Medium	Fixed
[M-05]	Monthly whale draw can be DoS'd over time (unbounded whaleMembers + O(n) selection) in `DrawCoordinator.sol` smart contract	Medium	Fixed
[M-06]	`PotController.sol#payoutWeekly()` function can be permanently DoS'd by a single recipient that can't receive tokens	Medium	Fixed
[M-07]	`FreeEntryVault.sol` smart contract authorization can brick weekly draw in VRF callback	Medium	Fixed
[L-01]	Missing `null` check causes `_executePurchase` to revert when `referralManager` is not set	Low	Fixed
[L-02]	Pending Jackpot claims, overwrites and locks out previous winner out of claims	Low	Fixed

ID	Title	Severity	Status
[L-03]	`EntryRouter.sol#setModules()` function allows zero-address modules and can brick purchases	Low	Fixed
[L-04]	Referral ETH transfer hard-reverts -> a non-payable referrer can permanently DoS buys	Low	Fixed
[L-05]	ETH refund hard-reverts -> some buyers (contracts) can't buy	Low	Fixed
[L-06]	Whale member list grows forever → monthly draw can become gas-DoS	Low	Fixed
[L-07]	BondsAdapter: rewards can get permanently stuck if `potController` is unset at claim time	Low	Fixed
[L-08]	`PotController.sol#handleYield()` function can revert and brick ticket purchases via `BondsAdapter.deposit()`	Low	Fixed
[L-09]	Second jackpot can brick weekly draw in `BondsAdapter.sol` smart contract while prior liquidation is pending	Low	Fixed
[I-01]	`EntryRouter.sol#initialize()` function doesn't validate module addresses (deployment misconfig -> broken system)	Informational	Fixed
[I-02]	`DrawCoordinator` setters allow zero addresses → easy admin misconfig DoS	Informational	Fixed
[I-03]	ReferralManager gifting behavior comment contradicts actual logic	Informational	Fixed
[I-04]	Jackpot USDC pot is effectively dead, making `claimJackpotYield()` misleading/no-op	Informational	Fixed
[U-01]	Streak manager user penalized and loses first bonus ticket	Unknown	Fixed

High

[H-01] Complete DOS of weekly draw if Jackpot winner cannot receive ETH

Description

If a jackpot winner is a smart contract that cannot or refuses to receive ETH, the `completeLiquidation()` function will permanently revert, causing `pendingLiquidation.completed` to remain false. This blocks all subsequent `requestLiquidation()` calls because of the `LiquidationAlreadyPending` check, preventing any new weekly draws from completing. This is possible because a contract that cannot receive ETH is still able to purchase tickets and become a winner (through purchasing with no refund, or by gifting to a address that cannot receive ETH)

This is not an immediate DOS of the protocol if the jackpot winner cannot receive ETH, the protocol will continue to function properly up until the point where there is a new jackpot winner. Once there is another jackpot winner, the protocol will be permanently DOS'd.

`completeLiquidation`:

```
/// @notice Complete pending liquidation when funds are received from
DexFi
/// @dev Called by owner/keeper when ETH arrives from bond liquidation (up
to 48 hours later)
function completeLiquidation() external payable onlyOwner nonReentrant {
    if (pendingLiquidation.amount == 0 || pendingLiquidation.completed) {
        revert NoPendingLiquidation();
    }

    require(msg.value >= pendingLiquidation.amount, "Insufficient ETH
received");

    pendingLiquidation.completed = true;
    address payable recipient = pendingLiquidation.recipient;
    uint256 amount = pendingLiquidation.amount;

    (bool ok, ) = recipient.call{value: amount}("");
 ->   require(ok, "Transfer failed");
```

Impact

- If that line fails and the jackpot winner cannot receive ETH (either malicious or on accident) the entire protocol becomes unusable in the following scenario:

1. The VRF callback initiates `requestRandomWords` and changes the state to where the callback must succeed (`round.drawing`) is set to true

- 2. `fulfillRandomWords` now MUST succeed for `round.drawing` to be set to false, or the initiation of the weekly drawing can never be called again
- 3. When another jackpot winner is selected and a user wins the jackpot, it will call `BondsAdapter::requestLiquidation` - which will FAIL because `pendingLiquidation.completed` will be FALSE and the amount will be > 0. This will prevent that function from executing and revert.
- 4. This reverts the entire VRF callback, leaving the protocol in a permanently DOS'd state

Recommendations

The strongest recommendation is to change the liquidation payment of the jackpot winner to a pull-pattern. Which will put the burden on the user to be able to accept the ETH payment if they choose to purchase tickets with a smart contract, but will not effect the protocol if they cannot.

Also, a different option would be to check to ensure that either only EOA accounts can purchase tickets, or smart contracts that purchase tickets are verified to be able to receive ETH.

Status

Fixed

[H-02] `BondsAdapter.sol#completeLiquidation()` requires `msg.value` (cannot pay from existing contract balance)

Description

`BondsAdapter.sol#completeLiquidation()` function requires `msg.value >= pendingLiquidation.amount`. If liquidation ETH is sent to the contract via the `receive()` function (common for async liquidation), that increases `address(this).balance` but doesn't set `msg.value` for `BondsAdapter.sol#completeLiquidation()`. This can block completion unless the owner re-sends ETH again in the call.

```
function completeLiquidation() external payable onlyOwner nonReentrant {
    if (pendingLiquidation.amount == 0 || pendingLiquidation.completed) {
        revert NoPendingLiquidation();
    }

    require(msg.value >= pendingLiquidation.amount, "Insufficient ETH 
received");

    pendingLiquidation.completed = true;
    address payable recipient = pendingLiquidation.recipient;
    uint256 amount = pendingLiquidation.amount;

    (bool ok, ) = recipient.call{value: amount}("");
    require(ok, "Transfer failed");

    emit LiquidationCompleted(recipient, amount);

    if (drawCoordinator != address(0)) {
        IDrawCoordinator(drawCoordinator).onBondsLiquidationComplete();
    }

    if (msg.value > amount) {
        (bool refundOk, ) = msg.sender.call{value: msg.value - amount}
("");
        require(refundOk, "Refund failed");
    }
}
```

Impact

- Jackpot bond liquidation completion can be blocked even when the contract already holds enough ETH from the liquidation. Funds can sit idle and the payout flow depends on the owner manually topping up ETH to pass the `msg.value` check.

Recommendations

Validate against contract balance instead of `msg.value`: `require(address(this).balance >= pendingLiquidation.amount, ...)`

Status

Fixed

[H-03] Permanent paid-purchase DoS in `EntryRouter.sol` smart contract if stored referrer cannot receive ETH

Description

Referral payout is done via a raw ETH call to `finalReferrer`. If that call fails, the entire purchase reverts. Since `ReferralManager.resolveReferrer()` stores the referrer permanently (`referrerOf[player]`), a buyer can get stuck with a referrer address that rejects ETH and every paid purchase will revert forever (until contracts are upgraded/migrated).

```
uint256 referralCut;
if (requiredEth > 0 && referralBps > 0 && finalReferrer != address(0)) {
    referralCut = (requiredEth * referralBps) / 10_000;
    if (referralCut > 0) {
        (bool okRef, ) = payable(finalReferrer).call{value: referralCut}
("");
        if (!okRef) revert ForwardFailed();
        emit ReferralPaid(finalReferrer, referralCut, drawId);
    }
}
```

- Permanent referrer binding:

```
function resolveReferrer(
    address player,
    address explicitReferrer,
    address giftRecipient
) external onlyAuthorized returns (address finalReferrer) {
    finalReferrer = referrerOf[player];
    if (finalReferrer != address(0)) {
        return finalReferrer;
    }
    address candidate = explicitReferrer;
    if (candidate == address(0) && giftRecipient != address(0)) {
        candidate = referrerOf[giftRecipient];
    }
    if (candidate != address(0) && candidate != player) {
        referrerOf[player] = candidate;
```

```
            ...
            finalReferrer = candidate;
        }
    }
```

Impact

- A buyer can be bricked from buying paid tickets if their stored referrer is (or becomes) a contract that rejects ETH. This is a real loss of functionality and can block revenue flow / user participation.

Recommendations

Don’t hard-revert purchases on referrer payout failure. Accrue referral rewards in a pull-based balance and let referrers claim.

Status

Fixed

[H-04] Ticket price USD decimals are wrong in EntryRouter smart contract -> tickets cost `~$0.002` instead of `$2` due to wrong USD decimals

Description

`EntryRouter.sol#initialize()` sets `ticketPriceUsd = 200_000` while the contract comments and logic treat `ticketPriceUsd` as USD with 8 decimals. With an 8-decimal ETH/USD Chainlink feed (the standard), this makes each ticket cost $0.002, not $2.00.

This wrong USD scale also flows into downstream accounting:

- `EntryRouter.sol#_executePurchase()` computes `usdVolumePaid = paidTickets * ticketPriceUsd`
- `DrawCoordinator.sol#recordWhaleVolume()` uses that value for whale eligibility tracking

```
// EntryRouter.sol#initialize()
ticketPriceUsd = 200_000;
```

```
// EntryRouter.sol#_requiredEth()
uint256 priceUint = uint256(answer);
uint256 numerator = ticketCount * ticketPriceUsd * 1e18;
requiredEth = (numerator + priceUint - 1) / priceUint;
```

Impact

- Users can buy tickets ~1000x cheaper than intended (major economic break). Referral volume and whale volume tracking are also scaled down by ~1000x, making whale eligibility and any USD-based analytics unreliable.

Recommendations

Set `ticketPriceUsd` to an actual 8-decimal USD value (e.g., `200_000_000` for $2.00) and keep all USD accounting in that same 8-decimal convention.

Status

Fixed

[H-05] Price feed decimals ignored -> mispriced ticket cost if feed decimals ≠ 8 in `EntryRouter.sol` smart contract

Description

`EntryRouter._requiredEth()` reads `priceFeedDecimals` but never uses it when converting USD price -> ETH. It assumes the Chainlink ETH/USD answer uses the same decimals as `ticketPriceUsd` (8 decimals). If the owner sets a feed with different decimals (or deploys with one), `requiredEth` becomes wrong by a power-of-10 factor.

```
function _requiredEth(uint256 ticketCount) internal view returns (uint256
requiredEth, int256 price, uint80 roundIdFeed) {
    if (ticketCount == 0) revert CountZero();
    (
        uint80 roundId,
        int256 answer,
        ,
        uint256 updatedAt,

    ) = priceFeed.latestRoundData();
    if (answer <= 0) revert BadPrice();
    if (updatedAt + 1 days <= block.timestamp) revert StalePrice();
    uint256 priceUint = uint256(answer);
    uint256 numerator = ticketCount * ticketPriceUsd * 1e18;
    requiredEth = (numerator + priceUint - 1) / priceUint;
    price = answer;
    roundIdFeed = roundId;
}
```

Impact

- If feed decimals ≠ 8, users can be overcharged or undercharged drastically (10x / 100x / 1e10 scale depending on decimals), causing direct financial loss or letting users buy tickets too cheaply.

Recommendations

Normalize the Chainlink answer to 8 decimals (or normalize `ticketPriceUsd` to the feed decimals) before computing `requiredEth`. Example: scale `priceUint` up/down using `priceFeedDecimals` so `ticketPriceUsd` and the oracle answer share the same decimal base.

Status

Fixed

[H-06] Weekly VRF callback in `DrawCoordinator.sol` smart contract can permanently brick the draw if it reverts (gas / external calls)

Description

`DrawCoordinator.sol#requestWeeklyDraw()` function flips `round.drawing = true`. The draw can only continue when the VRF callback succeeds and clears `drawing`. If the VRF callback reverts (out of gas due to large ticket count, or any downstream revert), the callback tx fails but `round.drawing` remains `true` from the earlier request tx. There is no timeout/reset path, so the round can be stuck forever.

Draw request sets `drawing = true`:

```
function requestWeeklyDraw() external whenNotPaused {
    Round storage round = rounds[currentDrawId];
    require(block.timestamp >= round.closeTime, "Draw not ready");
    require(!round.drawing && !round.drawn, "Already drawing");
    round.drawing = true;
    uint256 requestId = coordinator.requestRandomWords(
        vrfKeyHash,
        vrfSubscriptionId,
        vrfRequestConfirmations,
        vrfCallbackGasLimit,
        2
    );
    round.requestId = requestId;
    requestToDraw[requestId] = currentDrawId;
    emit DrawRequested(currentDrawId, requestId);
}
```

Callback does heavy unbounded work + external calls:

```
function fulfillRandomWords(uint256 requestId, uint256[] memory
randomWords) internal override {
    uint256 drawId = requestToDraw[requestId];
    Round storage round = rounds[drawId];
    require(round.drawing && !round.drawn, "Bad state");
    round.drawn = true;
    round.drawing = false;
    lastWeeklyRandomWord = randomWords[1];
    (uint8[5] memory winningMain, uint8 winningPB) =
_generateWinningNumbers(randomWords[0]);
    round.winningMainNumbers = winningMain;
    round.winningPowerball = winningPB;
    emit WinningNumbersDrawn(drawId, winningMain, winningPB);

    _processJackpotAndPowerballMatches(drawId, winningMain, winningPB);

    address payable[] memory winners = _selectWeeklyWinners(drawId,
randomWords[1]);
    if (winners.length > 0) {
        potController.payoutWeekly(winners);
    }
    emit DrawCompleted(drawId, round.ticketCount, _stripPayable(winners));
    dailyPoolSourceId = drawId;
    _openNextRound();
}
```

And inside `DrawCoordinator.sol#_processJackpotAndPowerballMatches()` function there is a full ticket loop and external calls:

```
function _processJackpotAndPowerballMatches(
    uint256 drawId,
    uint8[5] memory winningMain,
    uint8 winningPB
) internal {
    TicketRecord[] storage tickets_ = roundTickets[drawId];
    Round storage round = rounds[drawId];

    for (uint256 i = 0; i < tickets_.length; i++) {
        TicketRecord storage ticket = tickets_[i];

        uint8 mainMatches = _countMainNumberMatches(ticket.mainNumbers,
winningMain);
        bool powerballMatch = ticket.powerball == winningPB;

        if (mainMatches == 5 && powerballMatch) {
            if (!round.jackpotWon) {
                round.jackpotWon = true;
                round.jackpotWinner = ticket.player;
                emit JackpotWon(drawId, ticket.player, ticket.tokenId);

                _handleJackpotPayout(ticket.player, drawId);
            }
        } else {
            uint256 freeEntries = 0;
            ...
            if (freeEntries > 0 && address(freeEntryVault) != address(0))
{
```

```
                freeEntryVault.grantCredits(ticket.player, freeEntries);
                emit FreeEntryGranted(ticket.player, drawId);
            }
        }
    }
}
```

Status

Fixed

Medium

[M-01] Inconsistent Fee on Transfer Token Handling

Description

The `buyTicketsWithToken()` function does not account for fee-on-transfer (FOT) tokens when calculating the initial swap amount. If a user pays with an FOT token, the contract receives less tokens than `amountIn` due to transfer fees, but the swap function uses the original `amountIn value`. This causes the first swap to attempt using more tokens than the contract actually holds, leading to transaction failures or incorrect swap amounts.

```
    function buyTicketsWithToken(
        address token,
        uint256 amountIn,
        uint256 minEthOut,
        uint256 deadline,
        TicketIntent[] calldata tickets,
        address referrerHint,
        bool useCredits
    ) external whenNotPaused nonReentrant {
        TokenConfig memory cfg = tokenConfigs[token];
        if (!cfg.approved) revert TokenNotApproved();
        if (amountIn == 0) revert AmountZero();
        IERC20(token).safeTransferFrom(msg.sender, address(this),
amountIn);

->      uint256 receivedEth = _swapToEth(token, amountIn, minEthOut,
deadline, cfg);
        _executePurchase(msg.sender, receivedEth, tickets, referrerHint,
useCredits, token);
```

When a user purchases tickets with a fee-on-transfer token:

1. `IERC20(token).safeTransferFrom(msg.sender, address(this), amountIn);`
    - User transfers `amountIn` tokens
    - If token has transfer fee (for example: 1%), contract receives `amountIn * 0.99` tokens
2. `_swapToEth(token, amountIn, minEthOut, deadline, cfg);`
    - Passes original `amountIn` to swap function (not the actual received amount)
3. inside `_swapToEth`: `uint256 currentAmount = amountIn;`
    - Uses original `amountIn` instead of actual balance received
4. `IERC20(currentToken).safeIncreaseAllowance(segs[i].router, currentAmount);`
    - Approves `currentAmount` (original `amountIn`) to router
5. Swap functions called with `currentAmount`

- Router tries to transfer more tokens than contract actually holds

Result: Transaction reverts with insufficient balance error, or incorrect amounts are swapped and the accounting is wrong

The rest of the protocol has specific accounting logic to handle fee-on-transfer tokens, but in the example above it is not consistent.

Example of correct fee-on-transfer token handling: The code does correctly handle FOT tokens for intermediate swaps and output tokens:

```
uint256 balBefore = IERC20(outToken).balanceOf(address(this));
IUniswapV2Router02(segs[i].router).swapExactTokensForTokensSupportingFeeOn
TransferTokens(
    currentAmount,
    // ...
);
uint256 balAfter = IERC20(outToken).balanceOf(address(this));
-> currentAmount = balAfter - balBefore;      // Uses actual received
amount
```

```
uint256 balBeforeV3 = IERC20(outTokenV3).balanceOf(address(this));
IUniswapV3Router(segs[i].router).exactInput(params);
uint256 balAfterV3 = IERC20(outTokenV3).balanceOf(address(this));
-> currentAmount = balAfterV3 - balBeforeV3;      // Uses actual received
amount
```

Impact

- Despite having explicit fee on transfer token handling in the rest of the codebase, users wont be able to buy tickets or use fee on transfer tokens to buy tickets as their transaction will revert.

Recommendations

Check the actual balance received after transfer as such: By using the actual amount received

```
function buyTicketsWithToken(
    address token,
    uint256 amountIn,
    uint256 minEthOut,
    uint256 deadline,
    TicketIntent[] calldata tickets,
    address referrerHint,
    bool useCredits
) external whenNotPaused nonReentrant {
    TokenConfig memory cfg = tokenConfigs[token];
    if (!cfg.approved) revert TokenNotApproved();
    if (amountIn == 0) revert AmountZero();

->    uint256 balanceBefore = IERC20(token).balanceOf(address(this));
    IERC20(token).safeTransferFrom(msg.sender, address(this), amountIn);
    uint256 balanceAfter = IERC20(token).balanceOf(address(this));
->    uint256 actualAmount = balanceAfter - balanceBefore;       //
Account for FOT

->    uint256 receivedEth = _swapToEth(token, actualAmount, minEthOut,
deadline, cfg);
    _executePurchase(msg.sender, receivedEth, tickets, referrerHint,
useCredits, token);
}
```

Status

Fixed

[M-02] Whale volume counts free-credit tickets as paid volume in `EntryRouter.sol` smart contract

Description

Whale volume is recorded using total tickets requested (`tickets.length`) instead of paid tickets (`paidTickets`). Credits are consumed earlier and reduce how many tickets were actually paid for, but the whale counter still increases as if the user paid for all tickets.

```
uint256 creditsUsed = _consumeCredits(buyer, tickets.length, useCredits);
uint256 paidTickets = tickets.length - creditsUsed;
```

```
...
drawCoordinator.recordWhaleVolume(buyer, tickets.length * ticketPriceUsd);
```

Impact

- Users can inflate whale eligibility/whale stats using free credits, which breaks the intended economics/qualification rules for the monthly whale draw.

Recommendations

Record only paid volume:

- `drawCoordinator.recordWhaleVolume(buyer, paidTickets * ticketPriceUsd);`

If you intentionally want credits to count, document it explicitly and rename variables/events so it's not misleading.

Status

Fixed

[M-03] Whale status can be reached with free credits in `EntryRouter.sol` smart contract (not paid volume)

Description

`EntryRouter._executePurchase()` records whale volume using:

```
drawCoordinator.recordWhaleVolume(buyer, tickets.length * ticketPriceUsd);
```

This counts all tickets, including those covered by free entry credits.

But ETH payment is only required for:

```
paidTickets = tickets.length - creditsUsed;
requiredEth = _requiredEth(paidTickets);
```

So a user can inflate `usdVolume` without paying, by spending credits. That lets them cross `whaleThresholdUsd` and get appended into `whaleMembers`, becoming eligible for the monthly whale pot without actually being a whale (paid-volume-wise). If monthly pot has real yield backing it, this is real value extraction via eligibility bypass.

Simple flow:

- `_consumeCredits()` returns `creditsUsed`
- `paidTickets = tickets.length - creditsUsed`
- ETH required is correctly computed from `paidTickets`
- But whale volume is recorded using `tickets.length`, not `paidTickets`:

Recommendations

Record whale volume using paid tickets only:

```
drawCoordinator.recordWhaleVolume(buyer, paidTickets * ticketPriceUsd);
```

If the intended design is "activity volume" (paid + credits), document that explicitly and rename variables/events to avoid misinterpretation.

Status

Fixed

[M-04] Daily/Monthly draw winner can be biased (timestamp-based randomness + public trigger) in `DrawCoordinator.sol` smart contract

Description

`DrawCoordinator.sol#runDailyDraw()` and `DrawCoordinator.sol#_pickWhaleWinner()` derive "randomness" using `block.timestamp` and are publicly callable. That gives the caller (and especially a block producer / sequencer) influence over the output by choosing when the draw executes.

Daily:

```
function runDailyDraw() external whenNotPaused {
    require(lastWeeklyRandomWord != 0, "No randomness");
    require(block.timestamp >= nextDailyDrawTime, "Early");
    address[] storage pool = dailyPools[dailyPoolSourceId];
    require(pool.length > 0, "No pool");
    uint256 random = uint256(keccak256(abi.encode(lastWeeklyRandomWord,
dailyNonce++, pool.length, block.timestamp)));
    address winner = pool[random % pool.length];
    potController.payoutDaily(payable(winner));
    nextDailyDrawTime = block.timestamp + dailyInterval;
    emit DailyWinner(winner, dailyPoolSourceId);
}
```

Monthly:

```
function _pickWhaleWinner() internal returns (address) {
    uint256 len = whaleMembers.length;
    if (len == 0) {
        return address(0);
    }
    uint256 random = uint256(keccak256(abi.encode(lastWeeklyRandomWord,
monthlyNonce++, block.timestamp, len)));
    for (uint256 i = 0; i < len; i++) {
        uint256 idx = uint256(keccak256(abi.encode(random, i))) % len;
        address candidate = whaleMembers[idx];
        WhaleStats storage stats = whaleStats[candidate];
        if (stats.usdVolume >= whaleThresholdUsd && block.timestamp <=
stats.lastUpdated + 30 days) {
            return candidate;
        }
    }
    return address(0);
}
```

Impact

- Integrity/fairness failure: a motivated attacker can try to time the call (or use MEV / sequencer cooperation) to bias daily/monthly payouts toward a chosen address.

Recommendations

Remove `block.timestamp` from the randomness derivation. Maybe prefer a VRF request for daily/monthly as well or derive deterministically from VRF-only inputs (e.g., `keccak256(lastWeeklyRandomWord, nonce, drawEpochId)`). Make the outcome deterministic for a given epoch so the caller cannot influence it.

Status

Fixed

[M-05] Monthly whale draw can be DoS'd over time (unbounded whaleMembers + O(n) selection) in `DrawCoordinator.sol` smart contract

Description

`whaleMembers` is append-only: addresses are added once and never removed. `DrawCoordinator.sol#_pickWhaleWinner()` attempts up to `len` iterations and performs storage reads each time. Gas cost grows linearly with `whaleMembers.length`, eventually making monthly draws expensive and potentially uncallable.

Append-only list:

```
function recordWhaleVolume(address player, uint256 usdVolume) external
onlyEntryRouter {
    if (usdVolume == 0) return;
    WhaleStats storage stats = whaleStats[player];
    if (block.timestamp > stats.lastUpdated + 30 days) {
        stats.usdVolume = 0;
    }
    stats.usdVolume += uint96(usdVolume);
    stats.lastUpdated = uint64(block.timestamp);
    if (!stats.listed && stats.usdVolume >= whaleThresholdUsd) {
        stats.listed = true;
        whaleMembers.push(player);
    }
}
```

O(n) selection loop:

```
function _pickWhaleWinner() internal returns (address) {
    uint256 len = whaleMembers.length;
    if (len == 0) {
        return address(0);
    }
    uint256 random = uint256(keccak256(abi.encode(lastWeeklyRandomWord,
monthlyNonce++, block.timestamp, len)));
    for (uint256 i = 0; i < len; i++) {
        uint256 idx = uint256(keccak256(abi.encode(random, i))) % len;
        address candidate = whaleMembers[idx];
        WhaleStats storage stats = whaleStats[candidate];
        if (stats.usdVolume >= whaleThresholdUsd && block.timestamp <=
stats.lastUpdated + 30 days) {
            return candidate;
        }
    }
    return address(0);
}
```

Impact

- Monthly pot payout can become unreliable or stuck due to gas growth. At scale, draws may become too expensive to execute.

Recommendations

Maintain a compact active whale set (add/remove), not a forever-growing historical list.

Status

Fixed

[M-06] `PotController.sol#payoutWeekly()` function can be permanently DoS'd by a single recipient that can't receive tokens

Description

`payoutWeekly()` pushes USDC to every winner in a loop. If any single `safeTransfer()` reverts, the whole payout reverts and nobody gets paid.

```
function payoutWeekly(address payable[] calldata winners) external
onlyDrawCoordinator nonReentrant {
    uint256 len = winners.length;
    if (len == 0 || weeklyPot == 0) {
        return;
    }
    uint256 total = weeklyPot;
    uint256 share = total / len;
    uint256 payout = share * len;
    uint256 remainder = total - payout;
    weeklyPot = remainder;
    for (uint256 i = 0; i < len; i++) {
        payoutToken.safeTransfer(winners[i], share);
    }
    emit WeeklyPaid(_toArray(winners), share, remainder);
}
```

This is a classic "push payments" failure mode: one problematic recipient (blacklisted address, token restrictions, accidental `address(0)`, etc.) blocks payouts for everyone.

Impact

- Weekly payouts can be blocked indefinitely until the winners list is changed (or the bad recipient becomes transferable).
- Funds remain stuck in `weeklyPot` (since the function keeps reverting).

Recommendations

Switch to a pull-based payout model (claim pattern) or make the loop non-blocking. Record each winner's entitlement and let winners claim individually. If you must keep push payouts, use `try/catch` around each transfer and emit a "payout failed" event, while still completing the rest of the winners.

Status

Fixed

[M-07] `FreeEntryVault.sol` smart contract authorization can brick weekly draw in VRF callback

Description

`DrawCoordinator.sol#fulfillRandomWords()` function calls `_processJackpotAndPowerballMatches()` function. For partial matches it tries to grant free entries. `DrawCoordinator.sol#_processJackpotAndPowerballMatches()` calls `freeEntryVault.grantCredits(ticket.player, freeEntries)` when `freeEntries > 0` and `freeEntryVault != address(0)`.

But `FreeEntryVault.sol#grantCredits()` is protected by `onlyAuthorized`, which reverts if `authorizedCallers[msg.sender]` is false.

If `authorizedCallers[address(DrawCoordinator)] != true` (miswiring, vault swap, upgrade, or accidental de-authorization), `grantCredits()` reverts inside the VRF callback. That revert bubbles up and reverts the whole callback execution, so:

- `round.drawn = true` and `round.drawing = false` get reverted
- the round remains stuck at `drawing=true, drawn=false`
- Chainlink VRF will not "retry" the same callback automatically

This permanently bricks the weekly draw and blocks downstream progression (next round opening, winners payouts, daily/monthly randomness flow).

Recommendations

In `DrawCoordinator.sol#_processJackpotAndPowerballMatches()`, wrap the `grantCredits()` call in `try/catch` so the VRF callback cannot be bricked by `FreeEntryVault` authorization issues.

Status

Fixed

Low

[L-01] Missing `null` check causes `_executePurchase` to revert when `referralManager` is not set

Description

The _executePurchase() function will always revert if referralManager is set to address(0), even though the codebase treats referralManager as an optional module. The function calls referralManager.resolveReferrer() on line 531 without a null check, causing a low-level call failure. This contradicts the existing null check pattern used elsewhere in the codebase (line 534) and prevents the protocol from operating without a referral manager. The codebase has checks that referralManager is set before any action involving a call to it, which shows that it is a possibility that it may not be set and is optional, and if it is not set, it is not supposed to effect the EntryRouter logic, it will just be skipped, but this is not handled here, if it is not set, the entire buyTickets flow will always revert.

In other areas of the code, it makes calls to the referralManager only if it is not address(0), if it is not set, it skips it

```
-> if (usdVolumePaid > 0 && address(referralManager) != address(0)) {
    referralBps = referralManager.recordReferralPurchase(finalReferrer,
usdVolumePaid);
}
```

Again : *Location:** EntryRouter.sol:572

```
function _consumeCredits(address buyer, uint256 ticketCount, bool
useCredits) internal returns (uint256 used) {
    if (!useCredits || address(freeEntryVault) == address(0)) {  // ✅
Null check pattern
        return 0;
    }
    used = freeEntryVault.consumeCredits(buyer, ticketCount);
    // ...
}
```

Except here: There is no check that it is set, it makes an external call to referralManager regardless, which will ALWAYS revert if it is not set, DOS'ing the entire EntryRouter::buyTickets flow until it is set : Location: EntryRouter.sol:531

```
function _executePurchase(...) internal {
    // ... validation and setup ...

    address giftReferralSource = _giftReferralSource(tickets, buyer);
    -> address finalReferrer = referralManager.resolveReferrer(buyer,
referrerHint, giftReferralSource);
    // NO NULL CHECK - WILL REVERT IF address(0)

    uint16 referralBps;
    uint256 usdVolumePaid = paidTickets * ticketPriceUsd;
    -> if (usdVolumePaid > 0 && address(referralManager) != address(0)) {
// ✅ Has null check
        referralBps =
referralManager.recordReferralPurchase(finalReferrer, usdVolumePaid);
    }
    // ...
}
```

Impact

- If the `referralManager` is not set, most of the logic will be able to operate efficiently and effectively as if it was just an optional mechanism, but this failure to enforce this check during this logic will prevent users from ever purchasing tickets as `_executePurchase` will always revert. Rendering the protocol unable to sell tickets until `referralManager` is set. This ensures that the referral system is a must and can never be an optional mechanism.

Recommendations

Add the check that the rest of the protocol uses which allows the protocol to operate even if `referralManager` is not set.

Status

Fixed

[L-02] Pending Jackpot claims, overwrites and locks out previous winner out of claims

Description

The pendingJackpot struct is a single global variable that gets completely overwritten each time a new jackpot winner is selected. If a jackpot winner does not claim their yield (claimJackpotYield()) before a subsequent jackpot is won, they permanently lose access to their yield winnings. The yield claim function checks pendingJackpot.winner == msg.sender, which will fail once the struct is overwritten with a new winner's data. While bond liquidation is protected (a new jackpot cannot be won until the previous liquidation completes), yield claims are vulnerable because yield can be claimed immediately and there's no protection against a new jackpot being won after the previous winner's liquidation completes. This creates a permanent loss of funds scenario where winners who delay claiming yield can be locked out of their legitimate winnings.

This seems to be an optional or previously used mechanism, as the jackpot pot is reinvested for bond generation and the jackpot seems to be completely sent to the user via the `bondsAdapter` bond liquidation flow. But if this is ever used, this issue will persist.

```
699:721:contracts/DrawCoordinator.sol
function _handleJackpotPayout(address winner, uint256 drawId) internal {
    uint256 totalPrincipal = bondsAdapter.totalPrincipal();
    uint256 bondAmount = (totalPrincipal * 70) / 100;

    (uint256 yieldAmount,,,) = potController.getPots();

    pendingJackpot = PendingJackpot({
->        winner: winner,
        drawId: drawId,
        bondAmount: bondAmount,
        yieldAmount: yieldAmount,
        requestedAt: block.timestamp,
        bondsClaimed: false,
        yieldClaimed: false
    });

    if (bondAmount > 0) {
        bondsAdapter.requestLiquidation(bondAmount, payable(winner));
    }

    emit JackpotLiquidationRequested(drawId, winner, bondAmount,
yieldAmount);
}
```

The jackpot winner is stored in this struct. But is overridden by the new jackpot winner each time there is a new winner. So when the winner calls `claimJackpotYield`, THEY MUST be the current winner. If they didnt claim their rewards and a new jackpot winner is selected, they will be overridden and lose their reward here permanantely:

Impact

- The `pendingJackpot` struct is a single global variable that stores only one pending jackpot at a time. When a new jackpot is won, the entire struct is overwritten.

-
  ```
  ```
- 734:744:contracts/DrawCoordinator.sol
- /// @notice Allows jackpot winner to claim accumulated yield immediately
- /// @dev Yield in USDC can be claimed right away, bonds take up to 48 hours
- function claimJackpotYield() external { -> require(pendingJackpot.winner == msg.sender, "Not winner");
- require(!pendingJackpot.yieldClaimed, "Already claimed");
- pendingJackpot.yieldClaimed = true;
- potController.payoutJackpot(payable(msg.sender));
- emit JackpotYieldClaimed(pendingJackpot.drawId, msg.sender, pendingJackpot.yieldAmount);
- }

-
  ```
  ```
- If this jackpot payout flow is ever used, if a winner does not claim their rewards before a new jackpot winner is selected (as soon as the next draw) then they will permanently lose access to their payout and will be overridden by the new winner.

Recommendations

If this is planned to be used or kept as an optional mechanism for this protocol, the jackpot winner should be indexed and stored so that they remain entitled and able to pull their payout even if another jackpot winner is claimed in the next draws.

Status

Fixed

[L-03] `EntryRouter.sol#setModules()` function allows zero-address modules and can brick purchases

Description

`EntryRouter.sol#setModules()` function blindly sets module addresses. If any module is accidentally set to `address(0)` (or a wrong address), core flows revert later (minting, streak, credits, draw registration, bonds deposit).

```
function setModules(
    address fortuneFiTicket_,
    address referralManager_,
    address streakManager_,
    address freeEntryVault_,
    address drawCoordinator_,
    address bondsAdapter_
) external onlyOwner {
    fortuneFiTicket = FortuneFiTicket(fortuneFiTicket_);
    referralManager = ReferralManager(referralManager_);
    streakManager = StreakManager(streakManager_);
    freeEntryVault = FreeEntryVault(freeEntryVault_);
    drawCoordinator = DrawCoordinator(drawCoordinator_);
    bondsAdapter = BondsAdapter(payable(bondsAdapter_));
    emit ModulesUpdated();
}
```

Recommendations

Add zero-address checks for all module params (and optionally emit which module changed). Consider enforcing non-zero for required modules and allowing zero only for explicitly optional ones.

Status

Fixed

[L-04] Referral ETH transfer hard-reverts -> a non-payable referrer can permanently DoS buys

Description

If the resolved referrer is a contract that rejects ETH (no `receive()`/`fallback()` or intentionally reverts), purchases revert because referral payout uses a hard revert on failure.

```
if (requiredEth > 0 && referralBps > 0 && finalReferrer != address(0)) {
    referralCut = (requiredEth * referralBps) / 10_000;
    if (referralCut > 0) {
        (bool okRef, ) = payable(finalReferrer).call{value: referralCut}
("");
        if (!okRef) revert ForwardFailed();
        emit ReferralPaid(finalReferrer, referralCut, drawId);
    }
}
```

Impact

- A buyer can get "stuck" with a referrer that cannot receive ETH and then every future paid purchase reverts (since referrer is sticky via `ReferralManager.referrerOf`).

Recommendations

Don't hard-revert on referral payout failure. Instead: accrue unpaid referral balances for pull-claims or skip payout and emit an event for off-chain reconciliation.

Status

Fixed

[L-05] ETH refund hard-reverts -> some buyers (contracts) can't buy

Description

If `availableEth > requiredEth`, the router refunds the difference using `.call`. If the buyer is a contract that rejects ETH, the whole purchase reverts.

```
if (availableEth > requiredEth) {
    uint256 refund = availableEth - requiredEth;
    (bool ok, ) = payable(buyer).call{value: refund}("");
    if (!ok) revert RefundFailed();
}
```

Recommendations

Make refunds "pull-based" (store refundable balances), or allow the buyer to specify a refund address, or only refund when explicitly requested.

Status

Fixed

[L-06] Whale member list grows forever → monthly draw can become gas-DoS

Description

`whaleMembers` is append-only and never pruned. `_pickWhaleWinner()` loops up to `len`, and `runMonthlyDraw()` depends on it. Over time, this can become too expensive and revert.

```
if (!stats.listed && stats.usdVolume >= whaleThresholdUsd) {
    stats.listed = true;
    whaleMembers.push(player);
}
```

```
function _pickWhaleWinner() internal returns (address) {
    uint256 len = whaleMembers.length;
    if (len == 0) {
        return address(0);
    }
    uint256 random = uint256(keccak256(abi.encode(lastWeeklyRandomWord,
monthlyNonce++, block.timestamp, len)));
    for (uint256 i = 0; i < len; i++) {
        uint256 idx = uint256(keccak256(abi.encode(random, i))) % len;
        address candidate = whaleMembers[idx];
        WhaleStats storage stats = whaleStats[candidate];
        if (stats.usdVolume >= whaleThresholdUsd && block.timestamp <=
stats.lastUpdated + 30 days) {
            return candidate;
        }
    }
    return address(0);
}
```

Recommendations

Use an iterable set with pruning, or maintain a rolling “active whales” list per 30-day epoch or cap attempts and have a fallback selection strategy.

Status

Fixed

[L-07] BondsAdapter: rewards can get permanently stuck if `potController` is unset at claim time

Description

`_claimRewards()` forwards only the freshly harvested delta and only if `potController != address(0)`. If `potController` is unset when rewards are harvested, tokens remain in the adapter. Future claims compute `balanceAfter - balanceBefore`, so the old balance is never forwarded automatically.

```
IERC20 rewardToken = rewardPool.rewardToken();
uint256 balanceBefore = rewardToken.balanceOf(address(this));

rewardPool.deposit(0);

rewardAmount = rewardToken.balanceOf(address(this)) - balanceBefore;
emit RewardsClaimed(address(rewardToken), rewardAmount);

if (rewardAmount > 0 && potController != address(0)) {
    rewardToken.safeTransfer(potController, rewardAmount);
    IPotController(potController).handleYield(rewardAmount);
    emit YieldForwarded(rewardAmount);
}
```

Recommendations

If `potController` is unset, either revert (so rewards aren't harvested into a stuck state), or track/forward the full balance (including previously stuck tokens) when `potController` is later set.

Status

Fixed

[L-08] `PotController.sol#handleYield()` function can revert and brick ticket purchases via `BondsAdapter.deposit()`

Description

`PotController.handleYield()` does multiple external calls that can revert:

- transfers out of the `payoutToken` to `genesisWallet`
- transfers out of the `payoutToken` to `bondsAdapter`
- an external call into `bondsAdapter.reinvest()`

```
function handleYield(uint256 amount) external onlyBondsAdapter {
    if (amount == 0) {
        return;
    }
    uint256 jackpotShare = (amount * JACKPOT_BP) / BPS_DENOMINATOR;
    uint256 weeklyShare = (amount * WEEKLY_BP) / BPS_DENOMINATOR;
    uint256 genesisShare = (amount * GENESIS_BP) / BPS_DENOMINATOR;
    uint256 monthlyShare = (amount * MONTHLY_BP) / BPS_DENOMINATOR;
    uint256 dailyShare = amount - jackpotShare - weeklyShare -
genesisShare - monthlyShare;

    weeklyPot += weeklyShare;
    monthlyPot += monthlyShare;
    dailyPot += dailyShare;

    if (genesisShare > 0) {
        payoutToken.safeTransfer(genesisWallet, genesisShare);
    }

    if (jackpotShare > 0 && bondsAdapter != address(0)) {
        payoutToken.safeTransfer(bondsAdapter, jackpotShare);
        IBondsReinvestor(bondsAdapter).reinvest(jackpotShare);
    }

    emit PotsFunded(jackpotShare, weeklyShare, monthlyShare, dailyShare,
genesisShare);
}
```

This function is called from `BondsAdapter._claimRewards()`, which is called inside `BondsAdapter.deposit()`, which is called during ticket purchase. So any revert inside `handleYield()` bubbles up and can revert deposits / ticket purchases.

Impact

- If `genesisWallet` transfer fails (bad address / token restrictions), `handleYield()` reverts.
- If `bondsAdapter.reinvest()` reverts (misconfig, slippage, stale oracle, router issues, etc.), `handleYield()` reverts.
- Because this sits on the purchase path (`EntryRouter -> BondsAdapter.deposit -> _claimRewards -> PotController.handleYield`), ticket purchases can be blocked until the configuration / external dependency is fixed.

Recommendations

Make `handleYield()` resilient to failures in external calls so yield handling can't brick deposits/purchases - wrap `safeTransfer` to `genesisWallet` and the `reinvest()` leg in `try/catch` (or split "accounting" from "effects" and allow effects to fail without reverting the accounting).

Status

Fixed

[L-09] Second jackpot can brick weekly draw in `BondsAdapter.sol` smart contract while prior liquidation is pending

Description

The jackpot path calls `BondsAdapter.sol#requestLiquidation()` inside the VRF callback. BondsAdapter only allows one liquidation at a time; it reverts if another is pending. If a second jackpot occurs before the first liquidation is completed, the VRF callback will revert and the draw can become permanently stuck (same stuck-state as the callback-bricking issue, but this is a concrete, likely trigger).

DrawCoordinator calls liquidation inside callback path:

```
function _handleJackpotPayout(address winner, uint256 drawId) internal {
    uint256 totalPrincipal = bondsAdapter.totalPrincipal();
    uint256 bondAmount = (totalPrincipal * 70) / 100;

    (uint256 yieldAmount,,,) = potController.getPots();

    pendingJackpot = PendingJackpot({
        winner: winner,
        drawId: drawId,
        bondAmount: bondAmount,
        yieldAmount: yieldAmount,
        requestedAt: block.timestamp,
        bondsClaimed: false,
        yieldClaimed: false
    });

    if (bondAmount > 0) {
        bondsAdapter.requestLiquidation(bondAmount, payable(winner));
    }

    emit JackpotLiquidationRequested(drawId, winner, bondAmount,
yieldAmount);
}
```

BondsAdapter reverts if one is pending:

```
function requestLiquidation(uint256 amount, address payable recipient)
external onlyDrawCoordinator {
    if (pendingLiquidation.amount > 0 && !pendingLiquidation.completed) {
        revert LiquidationAlreadyPending();
    }
    if (amount > totalPrincipal) revert InsufficientPrincipal();

    totalPrincipal -= amount;

    pendingLiquidation = PendingLiquidation({
        recipient: recipient,
        amount: amount,
        requestedAt: block.timestamp,
        completed: false
    });

    emit LiquidationRequested(recipient, amount, block.timestamp);

    if (address(rewardPool) != address(0)) {
        _claimRewards();
        (uint256 stakedAmount, ) = rewardPool.userInfo(address(this));
        if (stakedAmount > 0) {
            rewardPool.withdraw(stakedAmount);
            emit BondsWithdrawn(address(rewardPool), stakedAmount);
        }
    }
}
```

Impact

- A second jackpot during the (up to) 48h liquidation window can revert the VRF callback and permanently freeze the weekly draw / round progression. This is a liveness failure with direct financial impact (payouts and protocol operation halted).

Recommendations

Move `BondsAdapter.sol#requestLiquidation()` out of the VRF callback. Have the callback only record the jackpot (winner/amounts), then trigger liquidation later via a separate keeper/owner tx that can retry or queue while a prior liquidation is pending.

Status

Fixed

Informational

[I-01] `EntryRouter.sol#initialize()` function doesn't validate module addresses (deployment misconfig -> broken system)

Description

`initialize()` validates `priceFeed_` and `weth`, but does not validate any of the module addresses. A bad deployment config can ship a bricked router.

```
function initialize(
    address owner_,
    address priceFeed_,
    address weth,
    address fortuneFiTicket_,
    address referralManager_,
    address streakManager_,
    address freeEntryVault_,
    address drawCoordinator_,
    address bondsAdapter_
) external initializer {
    if (priceFeed_ == address(0) || weth == address(0)) revert
ZeroAddress();
    __Ownable_init(owner_);
    __ReentrancyGuard_init();
    __Pausable_init();
    __UUPSUpgradeable_init();
    priceFeed = AggregatorV3Interface(priceFeed_);
    priceFeedDecimals = priceFeed.decimals();
    WETH = weth;
    fortuneFiTicket = FortuneFiTicket(fortuneFiTicket_);
    referralManager = ReferralManager(referralManager_);
    streakManager = StreakManager(streakManager_);
    freeEntryVault = FreeEntryVault(freeEntryVault_);
    drawCoordinator = DrawCoordinator(drawCoordinator_);
    bondsAdapter = BondsAdapter(payable(bondsAdapter_));
    ticketPriceUsd = 200_000;
}
```

Recommendations

Validate all required module addresses are non-zero at initialization.

Status

Fixed

[I-02] `DrawCoordinator` setters allow zero addresses → easy admin misconfig DoS

Description

Core module setters accept zero addresses with no validation. Setting any of these to `address(0)` breaks key flows (ticket registration, payouts, liquidation callbacks, credits).

```
function setEntryRouter(address newRouter) external onlyOwner {
    entryRouter = newRouter;
    emit EntryRouterUpdated(newRouter);
}

function setFreeEntryVault(address vault) external onlyOwner {
    freeEntryVault = FreeEntryVault(vault);
    emit FreeEntryVaultUpdated(vault);
}

function setPotController(address controller) external onlyOwner {
    potController = PotController(controller);
    emit PotControllerUpdated(controller);
}

function setBondsAdapter(address adapter) external onlyOwner {
    bondsAdapter = BondsAdapter(payable(adapter));
    emit BondsAdapterUpdated(adapter);
}
```

Recommendations

Require non-zero for required modules. If a module is intentionally optional, gate its usage consistently and emit explicit “disabled” events.

Status

Fixed

[I-03] ReferralManager gifting behavior comment contradicts actual logic

Description

The comment says “If gifting a ticket, the buyer becomes the recipient’s referrer.” The code does something different: it tries to use `referrerOf[giftRecipient]` (the gift recipient’s referrer), not the buyer.

```
/// @dev If player already has a referrer, returns existing one.
/// If gifting a ticket, the buyer becomes the recipient's referrer.
...
address candidate = explicitReferrer;
if (candidate == address(0) && giftRecipient != address(0)) {
    candidate = referrerOf[giftRecipient];
}
```

Recommendations

Fix the comment (or fix the implementation if the intended behavior is truly “buyer becomes referrer”).

Status

Fixed

[I-04] Jackpot USDC pot is effectively dead, making `claimJackpotYield()` misleading/no-op

Description

`handleYield()` never credits `jackpotPot`, yet jackpot-yield claiming calls `payoutJackpot()` which only pays `jackpotPot`. So “yield claim” is effectively always 0 under the current design.

```
function handleYield(uint256 amount) external onlyBondsAdapter {
    ...
    uint256 jackpotShare = (amount * JACKPOT_BP) / BPS_DENOMINATOR;
    ...
    weeklyPot += weeklyShare;
    monthlyPot += monthlyShare;
    dailyPot += dailyShare;

    ...
    payoutToken.safeTransfer(bondsAdapter, jackpotShare);
    IBondsReinvestor(bondsAdapter).reinvest(jackpotShare);
}
```

```
function payoutJackpot(address payable winner) external
onlyDrawCoordinator nonReentrant {
    uint256 amount = jackpotPot;
    if (amount == 0 || winner == address(0)) {
        return;
    }
    jackpotPot = 0;
    payoutToken.safeTransfer(winner, amount);
    emit JackpotPaid(winner, amount);
}
```

```
function claimJackpotYield() external {
    ...
    potController.payoutJackpot(payable(msg.sender));
    emit JackpotYieldClaimed(pendingJackpot.drawId, msg.sender,
pendingJackpot.yieldAmount);
}
```

Recommendations

Several mitigations:

- actually fund `jackpotPot`, or
- remove/rename `claimJackpotYield` and its events/fields to reflect reality (reinvest-only model), or
- have `claimJackpotYield` pull from a real yield balance tracked in PotController.

Status

Fixed

Unknown / Unclassified

[U-01] Streak manager user penalized and loses first bonus ticket

Description

The `newStreakManager.sol` contract has a bug in the bonus ticket calculation that prevents users from earning bonus tickets when they transition from streak 0 to streak 1. The subtraction logic (`bonusesAfter - bonusesBefore`) assumes bonuses were already calculated for existing tickets, but when tickets were purchased during streak 0 (where no bonuses are earned), this creates a "bonus debt" that prevents users from earning bonuses until they purchase enough tickets to overcome the incorrect `bonusesBefore` value.

Relevant code

```
function recordPaidTickets(address player, uint64 paidTicketCount)
external onlyAuthorized returns (uint32 newBonusEarned) {
    if (paidTicketCount == 0) return 0;

    StreakInfo storage info = streaks[player];

    if (info.current == 0) {
        info.paidTicketsThisStreak += paidTicketCount;
        emit PaidTicketsRecorded(player, paidTicketCount,
info.paidTicketsThisStreak, 0);
        return 0;
    }

    uint64 ticketsPerBonus = 53 - uint64(info.current);
    uint32 bonusesBefore = uint32(info.paidTicketsThisStreak /
ticketsPerBonus);
    info.paidTicketsThisStreak += paidTicketCount;
    uint32 bonusesAfter = uint32(info.paidTicketsThisStreak /
ticketsPerBonus);
    newBonusEarned = bonusesAfter - bonusesBefore;

    if (newBonusEarned > 0) {
        info.bonusTicketsEarned += newBonusEarned;
    }

    emit PaidTicketsRecorded(player, paidTicketCount,
  info.paidTicketsThisStreak, newBonusEarned);
}
```

The bug (subtraction logic prevents bonus earning)

- **Week 1 (first purchase – streak 0):** User buys 52 tickets. `recordPaidTickets(recipient, 52)` sets `paidTicketsThisStreak = 52` and returns 0 (no bonus while streak is 0).
- **Week 2 (consecutive – streak becomes 1):** User buys 1 ticket. `recordParticipation` sets `current = 1` but `paidTicketsThisStreak` is not reset (still 52). Then `recordPaidTickets(recipient, 1)` runs: `ticketsPerBonus = 52`, `bonusesBefore = 52 / 52 = 1`, `paidTicketsThisStreak = 53`, `bonusesAfter = 53 / 52 = 1`, so `newBonusEarned = 1 - 1 = 0`. The user should receive 1 bonus ticket (53 tickets at streak 1) but receives 0 because the subtraction assumes the 52 streak-0 tickets were already bonus-eligible.

Impact

Users transitioning from streak 0 to streak 1 never receive the bonus tickets they are owed for tickets bought at streak 0. They are permanently behind: e.g. with 105 total tickets at streak 1 they receive only 1 bonus (from the delta) instead of 2, and effectively need to purchase roughly twice the intended number of tickets to earn one bonus. This unfairly penalizes users and reduces the value of the streak mechanic.

Recommendations

Reset `paidTicketsThisStreak` when the streak increments (e.g. in `recordParticipation` when moving from streak 0 to 1), or change the bonus calculation so that tickets purchased during streak 0 are not treated as if they had already been counted toward bonuses. Ensure the first bonus at the new streak level is awarded correctly.

Status

Fixed

Conclusion

This security audit of the FortuneFi Perpetual Raffle identified 27 vulnerabilities across 6 High, 7 Medium, 9 Low, 4 Informational. All issues have been addressed by the development team.

Key findings addressed include:

- **High**: 6 high findings were identified and addressed
- **Medium**: 7 medium findings were identified and addressed
- **Low**: 9 low findings were identified and addressed
- **Informational**: 4 informational findings were identified and reported

Key improvements implemented include:

- Enhanced input validation and access controls
- Improved error handling and edge case coverage
- Strengthened security patterns and best practices

The development team's commitment to addressing these findings demonstrates a strong security-focused approach to smart contract development. The protocol has undergone significant improvements based on the audit findings, with all reported issues resolved. Continued monitoring and periodic security reviews are recommended as the protocol evolves.

This report was prepared by 33Audits & Co and represents our independent security assessment of the FortuneFi Perpetual Raffle project.