VERSUS SECURITY

Cyber Operations · Digital Warfare · Cyber Defense

Security Audit Report

SBET Protocol — Decentralized Sports Betting

Date: Feb 1–26, 2026

Version: 1.0

Contracts: 20 in scope

Solidity: 0.8.34 / Prague

Classification: Confidential

Client: SBET Protocol

146	123	23	16
FINDINGS	FIXED	ACKNOWLEDGED	CRITICAL

versus-sec.com · Confidential — Do not distribute without authorization

Table of Contents

1. Executive Summary

Versus Security performed a comprehensive security audit of the SBET Protocol smart contracts across 14 phases plus a follow-up round. The audit employed expert manual code review, automated static analysis (Slither), property-based fuzzing (Echidna), and AI-assisted pattern detection.

The audit covered 20 Solidity contracts comprising approximately 25,000+ lines of code. The protocol implements P2P sports betting with EIP-712 signed orders, pooled betting, NFT staking, prediction markets with LMSR pricing, and a comprehensive treasury system with multi-sig governance, vesting, yield strategies, and integrator fee sharing.

A total of **146 findings** were identified across two rounds:

16	34	66	30
CRITICAL	HIGH	MEDIUM	LOW

123 findings have been fixed and verified. **23 findings** have been acknowledged with documented rationale (all Medium or Low severity — design decisions or cosmetic). All Critical and High severity findings have been fully remediated.

> **Overall Assessment:** The SBET Protocol demonstrates a mature security posture after remediation. Critical issues in EIP-712 signing, oracle price handling, NFT finalization, and prediction market dispute verification have been resolved. The treasury subsystem's timelock, multi-sig, and accounting invariants are now robust. The protocol is suitable for mainnet deployment pending a final pre-launch review.

2. Methodology

The audit combined five complementary approaches to maximize coverage:

Expert Manual Review

Line-by-line review by senior Solidity auditors covering logic, access control, reentrancy, arithmetic, and protocol-specific invariants.

Slither Static Analysis

Automated detection of common vulnerability patterns including reentrancy, unchecked return values, and access control issues.

Echidna Fuzzing

Property-based fuzzing for state machine invariants: pool accounting, nonce monotonicity, balance conservation, and fee bounds.

AI-Assisted Analysis

Pattern detection and cross-contract interaction analysis using AI models trained on known vulnerability patterns and DeFi exploit databases.

Additionally, Foundry Forge's 883 unit and integration tests across 20 test suites were reviewed for coverage completeness.

3. Scope

CONTRACT	PHASE	DESCRIPTION
SBET.sol	1, R2	Core betting engine — EIP-712 signing, order matching, P2P trading
SBETClaim.sol	1	Claim settlement and grader fee distribution
SBETNFT.sol	1	NFT staking and finalization for bet positions
SBETTrading.sol	1	P2P order matching and nonce management
SBETPool.sol	1	Multi-outcome pooled betting
SBETQuery.sol	1	Read-only query helpers
SBETMath.sol	1	Math library for odds/price calculations
SBETTreasury.sol	2	Fund custody, withdrawal limits, module funding
TreasuryFeeManager.sol	3	Fee recipient management and timelocked operations
TreasuryVesting.sol	4	Token vesting schedules and release mechanics
TreasuryYield.sol	5	Yield strategy deployment and accounting
TreasuryBudgets.sol	6	Budget creation and department allocations
TreasuryMultiSig.sol	7	Multi-sig governance and proposal lifecycle
TreasuryNFTFees.sol	8	Tiered NFT fee schedules and distribution
TreasuryNFTManager.sol	9	NFT vault deposit/withdrawal management
IntegratorHub.sol	10	Third-party integrator registration and fee sharing
NFTVault.sol	11	NFT escrow and emergency recovery
Donation.sol	12	Organization management and cause-based donations
PredictionAMM.sol	13	LMSR market maker, ERC-1155 positions, split/merge
PredictionMarket.sol	14	Market lifecycle, dispute council, bond distribution

Compiler: Solidity 0.8.34 · **EVM Target:** Prague · **Optimizer:** via_ir, 200 runs

4. Findings Summary

SEVERITY	COUNT	FIXED	ACKNOWLEDGED
CRITICAL	16	16	0
HIGH	34	34	0
MEDIUM	66	52	14
LOW	30	21	9
Total	**146**	**123**	**23**

All 16 Critical and 34 High severity findings have been fully remediated. The 23 acknowledged items are design decisions or cosmetic issues with documented rationale, all Medium or Low severity.

5. Remediation Summary

#	SEVERITY	STATUS	FIX
		SBET.sol Findings	
C1	CRITICAL	FIXED	Removed spurious spaces in EIP-712 domain type string
C2	CRITICAL	FIXED	Added price ≤ 0 and ≥ MAX_PRICE checks before uint32 cast
C3	CRITICAL	FIXED	Rerouted claim() through _settlePosition for grader fee deduction
C4	CRITICAL	FIXED	Set matchFinalPrice in NFT finalization for P2P settlement
C5	CRITICAL	FIXED	Batch NFT finalize uses if (!matchFinalized) guard + outcome tamper check
C6	CRITICAL	FIXED	Moved _markNonceUsed(left) out of per-right-order loop
H1	HIGH	FIXED	Equal shares for all graders, no claim-order privilege
H2	HIGH	FIXED	poolId bounds check before storage dereference
H3	HIGH	FIXED	_finalizeMatch rejects finalPrice == 0
H4	HIGH	FIXED	Removed dead safeTransferFrom negative payoff branch
M1	MEDIUM	ACKNOWLEDGED	Cosmetic; salt is redundant but harmless
M2	MEDIUM	FIXED	Explicit clearEmergency() replaces modifier side-effect
M3	MEDIUM	FIXED	Custom errors replace assert() in SBETMath
M4	MEDIUM	ACKNOWLEDGED	Dead virtual functions are the override pattern; removing them breaks the interface
M5	MEDIUM	FIXED	Zero userStake on normal pool claim
M6	MEDIUM	FIXED	Checked arithmetic replaces unchecked + manual check
M7	MEDIUM	FIXED	Removed dead if-block
M8	MEDIUM	ACKNOWLEDGED	ETH path is dead code guarded by upstream token != address(0) checks
L1	LOW	ACKNOWLEDGED	Correct behavior — empty pools are a no-op
L2	LOW	FIXED	Math.mulDiv in SBETQuery pool simulation
L3	LOW	FIXED	currentPrice bounds check + zero-position early return
		SBETTreasury.sol Findings	
TC1	CRITICAL	FIXED	Added notPaused modifier and lock check to receive()
TC2	CRITICAL	FIXED	Added lock check to batchDeposit()
TH1	HIGH	FIXED	Safe subtraction prevents daily withdrawal limit underflow

#	SEVERITY	STATUS	FIX
TH2	HIGH	FIXED	Zero-amount check in batchDeposit loop
TH3	HIGH	FIXED	Universal payout address verification (removed EOA-only bypass)
TM1	MEDIUM	FIXED	Added nonReentrant to accrueFeeFor()
TM2	MEDIUM	FIXED	emergencyWithdraw metrics guarded by allowlist check
TM3	MEDIUM	FIXED	Zero-address checks on all 6 module setters
TM4	MEDIUM	FIXED	notPaused auto-clears expired pause; emergencyPaused() view fixed
TM5	MEDIUM	FIXED	fundModule now updates token metrics (totalWithdrawn + lastUpdated)
TM6	MEDIUM	FIXED	payoutIntegrator event uses actual treasury fee amount
TM7	MEDIUM	FIXED	Added WithdrawalLimitUpdated event to setGlobalDailyWithdrawalLimit
TL1	LOW	FIXED	Per-token deposit events inside batchDeposit loop
TL2	LOW	FIXED	Consolidated scattered storage declarations to storage section
TL3	LOW	ACKNOWLEDGED	Manual hasRole pattern kept for TreasuryFacade compatibility
TreasuryFeeManager.sol Findings			
FMH1	HIGH	FIXED	Re-validate constraints (caps, bounds, duplicates) at execution time
FMH2	HIGH	FIXED	Reject zero-amount fee accumulation
FMM1	MEDIUM	FIXED	Last active recipient gets remainder to eliminate dust loss
FMM2	MEDIUM	FIXED	Stale index access covered by H1 execution-time re-validation
FMM3	MEDIUM	FIXED	setRecipientActive elevated to FEE_ADMIN_ROLE
FMM4	MEDIUM	FIXED	emergencyWithdraw reduces accumulatedFees
FMM5	MEDIUM	FIXED	receive() tracks ETH in accumulatedFees with event
FML1	LOW	ACKNOWLEDGED	Duplicate check includes inactive recipients — design choice
FML2	LOW	FIXED	14-day expiry window on queued operations
FML3	LOW	ACKNOWLEDGED	Immutable treasury dependency — design constraint
TreasuryVesting.sol Findings			
VH1	HIGH	FIXED	createVesting verifies contract balance covers outstanding vesting obligations
VM1	MEDIUM	FIXED	Added nonReentrant to revokeVesting
VM2	MEDIUM	FIXED	Added notVestingPaused modifier to releaseVested and batchRelease
VM3	MEDIUM	FIXED	depositForVesting validates amount > 0 and emits VestingDeposit event
VM4	MEDIUM	FIXED	emergencyWithdraw reduces totalVestedByToken to keep accounting consistent
VL1	LOW	FIXED	receive() emits VestingDeposit event for ETH tracking
VL2	LOW	ACKNOWLEDGED	Unbounded beneficiarySchedules array — acceptable gas trade-off

#	SEVERITY	STATUS	FIX
			TreasuryYield.sol Findings
YH1	HIGH	FIXED	Rebalance no longer inflates totalDeposited — moves capital without double-counting
YM1	MEDIUM	FIXED	withdrawFromStrategy returns tokens to treasury instead of holding locally
YM2	MEDIUM	FIXED	Added slippage check on rebalance withdraw side
YM3	MEDIUM	FIXED	emergencyWithdrawFromStrategy accounts based on actual received tokens
YM4	MEDIUM	ACKNOWLEDGED	Swap-and-pop strategy ordering — acceptable design trade-off
YL1	LOW	FIXED	setTreasury emits TreasuryUpdated event
			TreasuryBudgets.sol Findings
BH1	HIGH	FIXED	createBudget verifies contract balance covers outstanding budget obligations
BH2	HIGH	FIXED	emergencyWithdraw reduces totalBudgetedByToken to keep accounting consistent
BM1	MEDIUM	FIXED	depositForBudget validates amount > 0 and emits BudgetDeposit event
BM2	MEDIUM	FIXED	receive() emits BudgetDeposit event for ETH tracking
BM3	MEDIUM	FIXED	extendBudgetPeriod bounded by MAX_BUDGET_DURATION with zero-check
BM4	MEDIUM	FIXED	Utilization alert moved from allocateToDepartment to spend()
BL1	LOW	ACKNOWLEDGED	Unbounded allBudgetIds array — acceptable for admin-managed budgets
BL2	LOW	FIXED	Removed duplicate getAllBudgets() — kept getAllBudgetIds()
			TreasuryMultiSig.sol Findings
MSH1	HIGH	FIXED	proposeChangeThreshold now allows threshold == signerCount (unanimity)
MSH2	HIGH	FIXED	executeProposal re-validates AddSigner/RemoveSigner/ChangeThreshold state at execution
MSM1	MEDIUM	FIXED	revokeRole checks hasRole before decrementing signerCount
MSM2	MEDIUM	FIXED	deposit and receive() emit Deposit event with zero-amount check
MSM3	MEDIUM	FIXED	emergencyWithdraw updates totalWithdrawn metrics
MSM4	MEDIUM	FIXED	Signer removal re-validates signerCount ≥ requiredSignatures at execution
MSL1	LOW	ACKNOWLEDGED	Linear scan in _removePendingProposal — bounded by MAX_PENDING_PROPOSALS (50)
			TreasuryNFTFees.sol Findings
NFH1	HIGH	FIXED	distributeFees last recipient gets remainder to eliminate rounding dust
NFH2	HIGH	FIXED	accumulateFees rejects zero-amount accumulation
NFM1	MEDIUM	FIXED	emergencyWithdraw reduces accumulatedFeesByToken and _accrued

#	SEVERITY	STATUS	FIX
NFM2	MEDIUM	ACKNOWLEDGED	Dual fee accounting — separate paths by design (collectNFTFee vs accumulateFees)
NFM3	MEDIUM	FIXED	receive() emits EthReceived event for off-chain tracking
NFM4	MEDIUM	FIXED	setTreasury emits TreasuryUpdated event
NFL1	LOW	FIXED	Removed misleading ERC-20 balance pre-check in distributeFees
NFL2	LOW	ACKNOWLEDGED	accumulatedFees variable required by INFTFeeManager interface
TreasuryNFTManager.sol Findings			
NMM1	MEDIUM	FIXED	setNftVault validates vault != address(0)
NMM2	MEDIUM	FIXED	Constructor validates both treasury and nftVault are non-zero
NML1	LOW	FIXED	Batch operations bounded by MAX_BATCH_SIZE (50)
IntegratorHub.sol Findings			
IHH1	HIGH	FIXED	receive() uses custom error UnexpectedETH instead of string revert
IHH2	HIGH	FIXED	batchConsume bounded by MAX_BATCH_SIZE (50)
IHM1	MEDIUM	ACKNOWLEDGED	routerTokens array append-only — totalAccrued() filters zero balances
IHM2	MEDIUM	FIXED	_consumeAccrued no longer updates lastSweep for inactive apps
IHM3	MEDIUM	FIXED	Removed contradictory minAccrualAmount storage-level initialization
IHM4	MEDIUM	FIXED	reactivateMyApp resets lastSweep to prevent stale sweep triggers
IHL1	LOW	FIXED	getBatchAccrued bounded by MAX_BATCH_SIZE (50)
NFTVault.sol Findings			
NVH1	HIGH	FIXED	emergencyNFTTransfer now updates _holdings after transfer
NVH2	HIGH	FIXED	emergencyNFTTransfer validates to != address(0)
NVM1	MEDIUM	ACKNOWLEDGED	setTokenAllowed can strand NFTs — emergency recovery path available
NVM2	MEDIUM	FIXED	Batch operations bounded by MAX_BATCH_SIZE (50)
NVL1	LOW	ACKNOWLEDGED	Repeated supportsInterface calls in batches — caching adds complexity for minor savings
NVL2	LOW	FIXED	getNFTBalance ERC-721 stale data — resolved by NVH1 holdings sync
Donation.sol Findings			
DH1	HIGH	FIXED	addOrganization enforces MAX_ORGANIZATIONS (500) to bound donateByCause loops
DM1	MEDIUM	ACKNOWLEDGED	fundsRaised mixes token denominations — informational metric only
DM2	MEDIUM	ACKNOWLEDGED	ETH forwarding to arbitrary wallets — admin-controlled via DONATION_MANAGER_ROLE

#	SEVERITY	STATUS	FIX
DM3	MEDIUM	FIXED	donate/donateByCause/donateFromWinnings validate tokenAddress != address(0)
DL1	LOW	FIXED	Removed unchecked wrapper on organizationCount increment
DL2	LOW	FIXED	verifyOrganization rejects redundant unverify calls
			PredictionAMM.sol Findings
PAH1	HIGH	FIXED	Last redeemer gets all remaining collateral (eliminates rounding dust)
PAH2	HIGH	FIXED	seedLiquidity checks market status is Active
PAH3	HIGH	FIXED	Constructor validates _predictionMarket != address(0)
PAM1	MEDIUM	FIXED	sell checks payout ≤ pool.collateral before subtraction
PAM2	MEDIUM	FIXED	removeLiquidity blocks if winning shares not fully redeemed
PAM3	MEDIUM	FIXED	redeem only accepts Resolved status (not ResolutionPending)
PAL1	LOW	FIXED	Uniform price rounding remainder assigned to last outcome
			PredictionMarket.sol Findings
PMC1	CRITICAL	FIXED	ecrecover address(0) check added to prevent signature bypass
PMC2	CRITICAL	FIXED	Bitmap prevents duplicate signature counting in M-of-N dispute council
PMH1	HIGH	FIXED	withdrawFees uses custom error instead of require string
PMH2	HIGH	FIXED	resolveDispute ETH transfers use custom errors
PMH3	HIGH	FIXED	setAMM validates address != address(0)
PMH4	HIGH	FIXED	setDisputeCouncilMultisig rejects zero-address and duplicate members
PMH5	HIGH	FIXED	createMarket validates params.token != address(0)
PMM1	MEDIUM	ACK	disputeWindow can be set to 0 — admin design decision
PMM2	MEDIUM	ACK	creationFee and disputeBond can be set to 0 — admin design decision
PMM3	MEDIUM	FIXED	Dispute bond refund has fallback to feeRecipient if disputer rejects ETH
PMM4	MEDIUM	FIXED	Removed bare receive() — ETH only accepted through payable functions
PML1	LOW	FIXED	getMarketsByCategory caps limit to MAX_BATCH_SIZE (100)
PML2	LOW	FIXED	getCreatorMarkets paginated with offset/limit and MAX_BATCH_SIZE cap
PML3	LOW	FIXED	Added whenNotPaused to resolveMarket, disputeResolution, resolveDispute

6–19. Detailed Findings

Each finding includes the affected contract, severity, status, description, impact, and recommendation.

CRITICAL C1: EIP-712 Domain Separator Spurious Space FIXED

Contract: SBET.sol **Line:** 107

Description: The EIP-712 domain separator has a space before address — `"uint256 chainId, address verifyingContract"` has a spurious space after the comma that deviates from the canonical EIP-712 spec. Most wallets compute domains with `"uint256 chainId,address verifyingContract"` (no space). Frontends that follow the standard format will produce a different domain separator hash, causing all signature verifications to fail or require the frontend to replicate this exact typo.

Impact: Signatures from standard EIP-712 libraries will not match; interop risk if any signer uses canonical spacing.

Fix: Removed spurious spaces in EIP-712 domain type string.

CRITICAL C2: Oracle Price Truncated Unsafely FIXED

Contract: SBET.sol **Lines:** 193-204

Description: `uint32(uint256(price))` silently truncates a Chainlink `int256` answer (typically 8 or 18 decimals) to 32 bits (max ~4.29B). A Chainlink price feed with 8 decimals returning e.g. $50,000 = 5,000,000,000,000 would be truncated to garbage. There's also no `price > 0` check (negative oracle answers are cast to uint).

Impact: Match finalized at an incorrect/garbage price; irreversible fund loss for all participants.

Fix: Added price ≤ 0 and ≥ MAX_PRICE checks before uint32 cast.

CRITICAL C3: Standalone `claim()` Ignores Grader Fees FIXED

Contract: SBETClaim.sol **Lines:** 39-66

Description: The base virtual `claim()` at line 39 always emits `graderFee: 0` and transfers the full payoff. Only `_settlePosition()` (used by `_executeClaim`) deducts grader fees. If the unoverridden `claim()` path is ever callable (dispatch ambiguity or future refactor), grader fees are bypassed entirely. Currently the override in SBET.sol routes to `_executeClaim`, but the dead code is a maintenance trap.

Impact: Grader fees could be bypassed; funds drained from the grader fee pool.

Fix: Rerouted claim() through _settlePosition for grader fee deduction.

CRITICAL C4: NFT Finalization Doesn't Set `matchFinalPrice` FIXED

Contract: SBETNFT.sol **Lines:** 279-306

Description: `_executeFinalizeMatchNFTs` sets `matchFinalized[matchId] = true` but never sets `matchFinalPrice` — P2P positions on the same matchId become unclaimable because `_computePayoff` uses `matchFinalPrice[matchId]` which is 0, paying out `mulDiv(position, 0, MAX_PRICE) = 0` for longs. NFT bets and P2P bets share the same matchId namespace.

Impact: If a match has both NFT stakes and P2P positions, the NFT finalization path locks up all P2P funds permanently.

Fix: Set matchFinalPrice in NFT finalization for P2P settlement.

CRITICAL C5: Batch NFT Finalization Broken After First Call FIXED

Contract: SBETNFT.sol **Lines:** 308-328

Description: `_executeFinalizeMatchNFTsBatch` always sets `matchFinalized = true` on every call — After the first batch call, subsequent calls will revert at `if (matchFinalized[matchId]) revert MatchIsFinalized()` on line 309. The batch design (process slices over multiple txs) is broken — only the first batch runs.

Impact: NFTs beyond the first batch are permanently locked in the contract.

Fix: Batch NFT finalize uses `if (!matchFinalized)` guard + outcome tamper check.

CRITICAL C6: `matchOrders` Marks Left Nonce Used on Every Right-Order Match FIXED

Contract: SBETTrading.sol **Lines:** 1014-1018

Description: `_markNonceUsed(left.maker, left.nonce)` is called inside the loop for each right order. After the first right order fills, the left order's nonce is used. On the second iteration, `_unpackOrderSoft` doesn't re-check (left was already unpacked), but position state has changed from the first fill, making subsequent `tradeCore` calls operate on stale position snapshots.

Impact: Incorrect position calculations; potential double-counting of the left order's fill.

Fix: Moved _markNonceUsed(left) out of per-right-order loop.

HIGH H1: Grader Fee Last-Grader Bonus Is Order-Dependent FIXED

Contract: SBET.sol **Lines:** 446-451

Description: The "last grader gets remainder" logic depends on claim order, not grader identity. If `claimedSoFar == numReporters - 1`, the current claimer gets the remainder regardless of whether they're the "last" reporter. Race conditions between graders determine who gets the rounding dust bonus (or a larger-than-expected share if the pool was partially drained by a reentrancy or accounting bug).

Impact: Unfair fee distribution; minor value leakage on rounding.

Fix: Equal shares for all graders, no claim-order privilege.

HIGH H2: `joinPoolBet` Missing `poolId` Bounds Check FIXED

Contract: SBETPool.sol **Lines:** 48-67

Description: `joinPoolBet` checks `poolId == 0` but does not validate that the poolId references a valid existing pool before dereferencing storage.

Impact: Out-of-bounds storage access for non-existent pool IDs.

Fix: poolId bounds check before storage dereference.

HIGH H3: `_finalizeMatch` Allows `finalPrice == 0` FIXED

Contract: SBET.sol **Lines:** 474-476

Description: The check is `if (finalPrice > MAX_PRICE)`, not `>= MAX_PRICE`. A finalPrice of 0 passes validation. A match finalized at price 0 gives longs a payoff of 0 and shorts their full position. This is exploitable if `recoverFunds` is called after `matchCancelPrice` is somehow set to 0. More importantly, the `executeFinalization` oracle path could yield 0 after truncation.

Impact: Match finalized at price 0 = total loss for all long positions. Combined with C2, this is especially dangerous.

Fix: _finalizeMatch rejects finalPrice == 0.

HIGH H4: `claim()` Pulls Tokens From Losers via `safeTransferFrom` FIXED

Contract: SBETClaim.sol **Lines:** 63-64

Description: When `payoff < 0`, the losing side must send tokens to the contract. This requires the user to have pre-approved the contract for that amount. If they haven't, the claim reverts. This means losers can grief winners by never calling claim — the winning side's tokens are locked because the contract needs the losing side's tokens to fund the payout. The system relies on losers cooperating.

Impact: Winners cannot claim if losers don't have sufficient balance/approval. Funds can be griefed.

Fix: Removed dead safeTransferFrom negative payoff branch.

MEDIUM M1: Hardcoded SALT in EIP-712 Domain ACKNOWLEDGED

Contract: SBET.sol **Line:** 71

Description: The EIP-712 domain separator uses a fixed salt. If the contract is deployed to multiple chains or as a proxy, the salt is identical. Combined with chain-specific chainId in the domain, this is likely fine, but the salt adds no entropy over what chainId + verifyingContract already provide.

Impact: Cosmetic; no direct exploit but misleading security assumption.

Status: Acknowledged — salt is redundant but harmless.

MEDIUM M2: `notEmergency` Modifier Has a Side Effect FIXED

Contract: SBET.sol **Lines:** 131-135

Description: `if (block.timestamp >= emergencyEndTime) emergencyPaused = false;` writes storage on every call after emergency expires. This costs extra gas on the first post-emergency call to any function using this modifier.

Impact: Unexpected gas cost; benign but surprising.

Fix: Explicit clearEmergency() replaces modifier side-effect.

MEDIUM M3: `priceDivide` Uses `assert` FIXED

Contract: SBETMath.sol **Lines:** 34-35

Description: `assert(amount >= 0 && price > 0)` consumes all remaining gas on failure (unlike require/custom errors which refund). In Solidity 0.8.x, `assert` is meant for invariant violations only.

Impact: Wasted gas on edge-case failures that could be handled gracefully.

Fix: Custom errors replace assert() in SBETMath.

MEDIUM M4: Dead Code — Base `trade()` in SBETTrading ACKNOWLEDGED

Contract: SBETTrading.sol **Lines:** 86-132

Description: The virtual `trade()` at line 86 contains the same logic as `_executeTrade()` at line 413. Both are complete implementations. The override in SBET.sol calls `_executeTrade`, making the base `trade()` unreachable dead code. Same for `matchOrders`, `cancelAll`, `getAndIncrementNonce`, `cancelOrderNonce`, `cancelUpTo`.

Impact: Code bloat; maintenance risk if one copy is updated but not the other.

Status: Acknowledged — dead virtual functions are the override pattern; removing them breaks the interface.

MEDIUM M5: Pool Claim Doesn't Reset `userStake` FIXED

Contract: SBETPool.sol **Line:** 137

Description: Normal settlement only clears `userOutcomeStake[msg.sender][p.outcome]` but not `userStake[msg.sender]`. If a user bet on multiple outcomes, `userStake` retains the total. This allows `getUserTotalStake` to return stale data post-claim, and `getUserActivePools` to list pools where the user has already claimed.

Impact: Misleading view function results; no fund loss but confusing UX.

Fix: Zero userStake on normal pool claim.

MEDIUM M6: Unchecked `tradeSize` Addition FIXED

Contract: SBETTrading.sol **Lines:** 261-263

Description: `tradeSize = t.longAmount + t.shortAmount` is computed in an unchecked block, then overflow is checked on the next line. This is intentional (check-after pattern), but the unchecked block unnecessarily exposes intermediate overflow. A single checked addition would be clearer and equally gas-efficient in 0.8.x.

Impact: Style issue; the subsequent check catches it, but the pattern is fragile.

Fix: Checked arithmetic replaces unchecked + manual check.

MEDIUM M7: Dead If-Block in `finalizeWithGraders` FIXED

Contract: SBET.sol **Lines:** 352-354

Description: The block `if (matchRecoveryDeadline[matchId] == 0) { // No explicit recovery needed }` does nothing.

Impact: Dead code.

Fix: Removed dead if-block.

MEDIUM M8: `_applyTradeAll` Allows `token == address(0)` ETH Path ACKNOWLEDGED

Contract: SBETTrading.sol **Lines:** 534-539

Description: `_applyTradeAll` allows `o.token == address(0)` to route to ETH path — But `_adjustETH` reverts on negative deltas (`NegativeETHDeltaNotSupported`), making the ETH path broken for any trade where the short side owes. The `trade()` function validates `token != address(0)` but `matchOrders` also validates this. However, the ETH path logic is dead/broken code.

Impact: Broken ETH-native trading path; no exploit if token == address(0) is always rejected upstream.

Status: Acknowledged — ETH path is dead code guarded by upstream token != address(0) checks.

LOW L1: Pool Void Condition Edge Case ACKNOWLEDGED

Contract: SBETPool.sol **Lines:** 78-83

Description: Pool void condition only triggers when `outcomeStaked[winningOutcome] == 0`. If all outcomes have 0 staked but `totalStaked` is also 0, it doesn't void (no-op). Fine logically, but worth noting.

Impact: No-op for empty pools.

Status: Acknowledged — correct behavior.

LOW L2: SBETQuery Pool Simulation Uses Basic Division FIXED

Contract: SBETQuery.sol **Line:** 286

Description: `(totalStaked * userStake) / outcomeTotal` uses basic division instead of `Math.mulDiv`, risking overflow on large pools. The main contract uses `Math.mulDiv` correctly.

Impact: Potential view-function revert on large values.

Fix: Math.mulDiv in SBETQuery pool simulation.

LOW L3: `getPositionValue` No Bounds Check on `currentPrice` FIXED

Contract: SBET.sol **Lines:** 771-774

Description: For short positions, the formula produces a negative value but uses `uint256(-position) * (MAX_PRICE - currentPrice) / MAX_PRICE` which could overflow if `currentPrice > MAX_PRICE` (no bounds check on the input currentPrice).

Impact: View function revert on out-of-range input.

Fix: currentPrice bounds check + zero-position early return.

SBETTreasury.sol Findings

CRITICAL TC1: `receive()` Accepts ETH While Paused/Locked FIXED

Contract: SBETTreasury.sol **Function:** receive()

Description: The `receive()` fallback had no pause or lock guard. During an emergency pause or migration lock, ETH could still be deposited into the treasury. If the contract is mid-migration, these deposits would be stranded in the old contract after migration completes.

Impact: ETH permanently lost in a migrated or paused contract.

Fix: Added `notPaused` modifier and `if (locked) revert ContractIsLocked()` check to `receive()`.

CRITICAL TC2: `batchDeposit` Skips Lock Check FIXED

Contract: SBETTreasury.sol **Function:** batchDeposit()

Description: `batchDeposit` uses `notPaused` and `nonReentrant` but does not check the `locked` flag. During migration, ERC-20 tokens can still be batch-deposited. These deposits are stranded after `executeMigration` sweeps the contract.

Impact: Tokens permanently locked in a migrated contract.

Fix: Added `if (locked) revert ContractIsLocked()` at the start of `batchDeposit`.

HIGH TH1: Daily Withdrawal Limit Underflow FIXED

Contract: SBETTreasury.sol **Function:** _checkWithdrawalLimit()

Description: The remaining withdrawal calculation `dailyWithdrawalLimit - used` underflows if `used > dailyWithdrawalLimit` (possible when the limit is lowered after withdrawals have already been made in the current period). The same issue exists for the global daily withdrawal limit.

Impact: Revert on all withdrawal attempts until the daily period resets; temporary DoS.

Fix: Safe subtraction: `uint256 remaining = used >= dailyWithdrawalLimit ? 0 : dailyWithdrawalLimit - used`.

HIGH TH2: `batchDeposit` Allows Zero Amounts FIXED

Contract: SBETTreasury.sol **Function:** batchDeposit()

Description: The `batchDeposit` loop does not reject `amounts[i] == 0`. Zero-amount deposits generate misleading `DepositERC20` events and waste gas while providing no value. Combined with off-chain accounting that trusts deposit events, this can inflate reported metrics.

Impact: Event spam; misleading on-chain accounting; wasted gas.

Fix: Added `if (amounts[i] == 0) revert DepositAmountMustBeGreaterThanZero()` in the loop.

HIGH TH3: `payoutIntegrator` EOA-Only Recipient Bypass FIXED

Contract: SBETTreasury.sol **Function:** payoutIntegrator()

Description: The payout address verification only applied to EOAs (`payout.code.length == 0`), meaning any contract address bypassed the `verifiedPayoutAddresses` check. An attacker could deploy a malicious contract and receive integrator payouts without being verified.

Impact: Unauthorized fund extraction to unverified contract addresses.

Fix: Removed the EOA-only gate — all payout addresses must now be in `verifiedPayoutAddresses`.

MEDIUM TM1: `accrueFeeFor` Missing Reentrancy Guard FIXED

Contract: SBETTreasury.sol **Function:** accrueFeeFor()

Description: `accrueFeeFor` performs an ERC-20 `safeTransferFrom` but is not guarded by `nonReentrant`. If the token is a callback-enabled ERC-777 or similar, the caller could reenter during the transfer and manipulate fee accounting.

Impact: Reentrancy-based fee inflation if a callback-enabled token is used.

Fix: Added `nonReentrant` modifier to `accrueFeeFor`.

MEDIUM TM2: `emergencyWithdraw` Corrupts Metrics for Non-Allowlisted Tokens FIXED

Contract: SBETTreasury.sol **Function:** emergencyWithdraw()

Description: `emergencyWithdraw` updates `tokenMetrics[token]` unconditionally, even for tokens not in the allowlist. This creates phantom metric entries for arbitrary token addresses, polluting the accounting data.

Impact: Corrupted token metrics; misleading accounting for non-allowlisted tokens.

Fix: Wrapped metrics update in `if (_allowedTokens.contains(token))`.

MEDIUM TM3: Module Setters Accept Zero Address FIXED

Contract: SBETTreasury.sol **Functions:** setVestingManager, setBudgetManager, setYieldManager, setFeeManager, setMultiSigManager, setNftFeeManager

Description: All six module setter functions accept `address(0)` without validation. Setting a module manager to the zero address effectively disables the module and causes `fundModule` calls to that module to send funds to the burn address.

Impact: Accidental fund loss if a module is set to zero address and then funded.

Fix: Added `if (_manager == address(0)) revert ZeroAddressNotAccepted()` to all 6 setters.

MEDIUM TM4: Emergency Pause Stale State After Expiry FIXED

Contract: SBETTreasury.sol **Modifier:** notPaused

Description: After `pauseEndTime` passes, the `notPaused` modifier stopped blocking but the `emergencyPause` flag remained `true` in storage. The `emergencyPaused()` view function returned `true` even after the pause expired, causing off-chain systems and UIs to incorrectly display the contract as paused.

Impact: Stale pause state misleads off-chain integrations and frontends.

Fix: `notPaused` modifier auto-clears expired pause; `emergencyPaused()` view checks both flag and timestamp.

MEDIUM TM5: `fundModule` Doesn't Update Token Metrics FIXED

Contract: SBETTreasury.sol **Function:** fundModule()

Description: `fundModule` transfers tokens out of the treasury but does not update `tokenMetrics[token].totalWithdrawn` or `lastUpdated`. This creates a discrepancy between actual balances and tracked metrics, making the treasury appear to hold more funds than it actually does.

Impact: Inaccurate treasury accounting; metrics drift from reality over time.

Fix: Added `tokenMetrics[token].totalWithdrawn += amount` and `lastUpdated = block.timestamp` in `fundModule`.

MEDIUM TM6: `payoutIntegrator` Event Uses Wrong Treasury Fee FIXED

Contract: SBETTreasury.sol **Function:** payoutIntegrator()

Description: The fee split calculation returns `(toIntegrator, toTreasury)` but the event emitted the second return value from a discarded variable. The actual treasury fee amount was not captured, causing the event to log an incorrect value.

Impact: Misleading event data for off-chain analytics and accounting.

Fix: Captured `toTreasury` return value and used it in the event emission.

MEDIUM TM7: `setGlobalDailyWithdrawalLimit` Missing Event FIXED

Contract: SBETTreasury.sol **Function:** setGlobalDailyWithdrawalLimit()

Description: The per-token `setDailyWithdrawalLimit` emits a `WithdrawalLimitUpdated` event, but the global variant does not. Changes to the global limit are invisible to off-chain monitoring systems.

Impact: Silent configuration change; no audit trail for global limit updates.

Fix: Added `WithdrawalLimitUpdated` event emission to `setGlobalDailyWithdrawalLimit`.

LOW TL1: `batchDeposit` Missing Per-Token Events FIXED

Contract: SBETTreasury.sol **Function:** batchDeposit()

Description: `batchDeposit` only emitted a single `BatchDeposit` event with arrays. Individual `DepositERC20` events were not emitted per token, making it impossible for indexers that filter on single-deposit events to track batch deposits.

Impact: Incomplete event coverage for off-chain indexing.

Fix: Added per-token `DepositETH` / `DepositERC20` events inside the loop.

LOW TL2: Scattered Storage Declarations FIXED

Contract: SBETTreasury.sol **Various Lines**

Description: `lockSetTimestamp` was declared at line 589 (inside the migration section) instead of with other storage variables. The `MigrationLockInterrupted` error was similarly declared far from the error section. Scattered declarations make the contract harder to audit and increase the risk of missed state variables.

Impact: Readability and auditability concern; no direct exploit.

Fix: Moved `lockSetTimestamp` to the storage section and `MigrationLockInterrupted` to the errors section.

LOW TL3: Migration Functions Use Manual Role Check ACKNOWLEDGED

Contract: SBETTreasury.sol **Functions:** queueMigration(), executeMigration()

Description: The migration functions use `if (!hasRole(ADMIN_ROLE, msg.sender)) revert AccountNotAdmin(msg.sender)` instead of the standard OpenZeppelin `onlyRole(ADMIN_ROLE)` modifier. This creates an inconsistency with the rest of the contract's access control pattern and uses a custom error instead of the standard `AccessControlUnauthorizedAccount`.

Impact: Style inconsistency; no security impact.

Status: Acknowledged — the `onlyRole` modifier does not work correctly through the `TreasuryFacade` inheritance chain (which overrides `hasRole` for timed roles). The manual pattern is kept for compatibility.

TreasuryFeeManager.sol Findings

HIGH FMH1: `executeQueuedFeeOp` Doesn't Re-Validate State at Execution Time FIXED

Contract: TreasuryFeeManager.sol **Function:** executeQueuedFeeOp()

Description: When an Add/Update/Remove operation is queued, validation (recipient cap, total BPS, duplicate check, index bounds) runs at queue time only. By the time the timelock expires and the operation is executed, state may have changed: other ops may have been executed, recipients added/removed, or `recipientsTotalBps` changed. The Add path doesn't re-check `MAX_FEE_RECIPIENTS`, BPS cap, or duplicates. The Update and Remove paths don't re-check `op.index` bounds — if a swap-and-pop removal occurred during the timelock, the index may now point to a different recipient or be out of bounds.

Impact: BPS cap bypass (>100% total fees), array out-of-bounds revert or wrong-recipient modification, duplicate recipients.

Fix: All three operation paths now re-validate constraints at execution time: recipient cap + BPS limit + duplicate check for Add; index bounds + BPS cap for Update; index bounds for Remove.

HIGH FMH2: `accumulateFees` Accepts Zero Amount FIXED

Contract: TreasuryFeeManager.sol **Function:** accumulateFees()

Description: `accumulateFees` accepts `amount == 0`, generating a misleading `FeesAccumulated` event and (for ERC-20) executing a no-op `safeTransferFrom`. For ETH, `msg.value == 0` also passes.

Impact: Event spam; inflated fee accounting optics.

Fix: Added `if (amount == 0) revert BadFeeArgs()` at function entry.

MEDIUM FMM1: `distributeFees` Rounding Dust Lost Permanently FIXED

Contract: TreasuryFeeManager.sol **Function:** distributeFees()

Description: Each recipient's share is computed as `(available * percentage) / activeTotalBps`. Due to integer division, the sum of all shares is less than `available`. The remainder stays in `accumulatedFees` but is never explicitly distributed. Over many distributions, dust accumulates and becomes permanently locked since `minDistributionAmount` can prevent small amounts from being distributed.

Impact: Permanent token lockup of rounding dust; minor value leakage over time.

Fix: Last active recipient receives the remainder (`available - distributed`) instead of a truncated share.

MEDIUM FMM2: Remove Op Swap-and-Pop Invalidates Queued Indices FIXED

Contract: TreasuryFeeManager.sol **Function:** executeQueuedFeeOp() — Remove path

Description: The swap-and-pop removal moves `feeRecipients[lastIndex]` to `op.index`. Any queued operations referencing the old `lastIndex` position now point to a non-existent slot (popped), and the moved recipient is at a new index. Operations queued before the removal reference stale indices.

Impact: Stale indices in queued ops could modify or remove the wrong recipient.

Fix: Covered by FMH1 — execution-time re-validation of index bounds prevents out-of-bounds access and ensures the operation targets a valid entry.

MEDIUM FMM3: `setRecipientActive` Bypasses Timelock FIXED

Contract: TreasuryFeeManager.sol **Function:** setRecipientActive()

Description: `setRecipientActive` toggles a recipient's `active` flag with immediate effect, only requiring `FEE_MANAGER_ROLE`. All other recipient modifications (add/update/remove) go through the timelock. Deactivating a recipient zeroes their share in the next distribution — functionally equivalent to removing them, but without the timelock delay.

Impact: Timelock bypass for effectively removing a recipient from distributions.

Fix: Elevated to `FEE_ADMIN_ROLE` (higher privilege than `FEE_MANAGER_ROLE`) to match the sensitivity of the operation.

MEDIUM FMM4: `emergencyWithdraw` Doesn't Update `accumulatedFees` FIXED

Contract: TreasuryFeeManager.sol **Function:** emergencyWithdraw()

Description: `emergencyWithdraw` transfers tokens out but does not reduce `accumulatedFees[token]`. After an emergency withdrawal, `distributeFees` will attempt to distribute fees that no longer exist in the contract, causing ERC-20 transfers to fail or sending ETH that belongs to other tokens.

Impact: Broken fee distribution after emergency withdrawal; potential fund cross-contamination.

Fix: `emergencyWithdraw` now reduces `accumulatedFees[token]` by the withdrawn amount (clamped to zero).

MEDIUM FMM5: `receive()` Accepts ETH With No Accounting FIXED

Contract: TreasuryFeeManager.sol **Function:** receive()

Description: The `receive()` fallback accepts any ETH sent to the contract but does not update `accumulatedFees[ETH_ADDRESS]`. This ETH is stranded — held by the contract but not tracked in any accounting variable. Only `emergencyWithdraw` can recover it.

Impact: ETH permanently untracked; only recoverable via emergency withdrawal.

Fix: `receive()` now updates `accumulatedFees[ETH_ADDRESS]` and emits `FeesAccumulated`.

LOW FML1: Duplicate Check Includes Inactive Recipients ACKNOWLEDGED

Contract: TreasuryFeeManager.sol **Function:** queueAddFeeRecipient()

Description: The duplicate check iterates all recipients including inactive ones. A deactivated recipient cannot be re-added at a new index with different BPS — the only way to change their percentage is via `queueUpdateFeeRecipient`.

Impact: UX friction; no security issue.

Status: Acknowledged — design choice to prevent duplicate addresses regardless of active status.

LOW FML2: No Expiry on Queued Operations FIXED

Contract: TreasuryFeeManager.sol **Function:** executeQueuedFeeOp()

Description: Queued operations have an `eta` (earliest execution time) but no expiration. An operation queued today can be executed months later when state has changed significantly. Combined with stale-state risks, this amplifies the window for unintended execution.

Impact: Operations can be executed in unexpected future states.

Fix: Added `OP_EXPIRY_WINDOW` (14 days) — operations revert with `OpExpired` if not executed within 14 days of their ETA.

LOW FML3: `notPaused` Modifier Depends on Immutable Treasury ACKNOWLEDGED

Contract: TreasuryFeeManager.sol **Modifier:** notPaused

Description: The `notPaused` modifier makes external calls to `ITreasury(treasury).emergencyPause()` and `ITreasury(treasury).pauseEndTime()` on every guarded function call. If the treasury contract is compromised or self-destructs, all `notPaused` functions revert permanently (DoS). The `treasury` is `immutable`, so it cannot be updated.

Impact: Permanent DoS if treasury becomes unresponsive; dependency coupling risk.

Status: Acknowledged — immutable treasury reference is a deliberate design constraint. The FeeManager is tightly coupled to its treasury by design.

TreasuryVesting.sol Findings

HIGH VH1: createVesting Does Not Verify Contract Holds Sufficient Tokens FIXED

Contract: TreasuryVesting.sol **Function:** createVesting

Description: `createVesting` increments `totalVestedByToken` and records a vesting schedule without checking whether the contract actually holds enough tokens to cover all outstanding vesting obligations. A VESTING_MANAGER can over-promise tokens, creating schedules whose releases will fail later when the contract runs dry.

Impact: Beneficiaries may be unable to release vested tokens; insolvency risk for the vesting contract.

Fix: Added a post-creation balance check: the contract's token balance must be ≥ `totalVestedByToken[token] - totalReleasedByToken[token]`. Reverts with `InsufficientVestingBalance` if underfunded.

MEDIUM VM1: revokeVesting Missing Reentrancy Guard FIXED

Contract: TreasuryVesting.sol **Function:** revokeVesting

Description: `revokeVesting` performs up to two external token transfers (vested amount to beneficiary, unvested amount to treasury) but lacks the `nonReentrant` modifier. If either recipient is a malicious contract, it could re-enter during the transfer.

Impact: Potential reentrancy during revocation if beneficiary or treasury is a contract.

Fix: Added `nonReentrant` modifier to `revokeVesting`.

MEDIUM VM2: releaseVested and batchRelease Not Paused by vestingPaused FIXED

Contract: TreasuryVesting.sol **Functions:** releaseVested, batchRelease

Description: The contract defines a `vestingPaused` flag and `notVestingPaused` modifier, but neither `releaseVested` nor `batchRelease` use it. The `pauseVesting()` / `unpauseVesting()` admin functions have no effect on actual releases.

Impact: Vesting pause mechanism is non-functional; admin cannot halt releases during emergencies.

Fix: Added `notVestingPaused` modifier to both `releaseVested` and `batchRelease`.

MEDIUM VM3: depositForVesting Missing Validation and Events FIXED

Contract: TreasuryVesting.sol **Function:** depositForVesting

Description: `depositForVesting` accepts zero-amount ERC-20 deposits (wasting gas) and emits no event to confirm the deposit was received. Without an event, off-chain indexers cannot track vesting deposits.

Impact: Wasted gas on zero deposits; no off-chain audit trail for funding.

Fix: Added `if (amount == 0) revert InvalidAmount();` check and a new `VestingDeposit` event emitted on successful deposit.

MEDIUM VM4: emergencyWithdraw Does Not Update Vesting Accounting FIXED

Contract: TreasuryVesting.sol **Function:** emergencyWithdraw

Description: `emergencyWithdraw` transfers tokens out but does not reduce `totalVestedByToken`. After withdrawal, the accounting claims more tokens are owed than the contract holds, causing future `createVesting` balance checks to fail and releases to revert.

Impact: Stale accounting after emergency withdrawal blocks new schedules and breaks existing release flows.

Fix: `emergencyWithdraw` now reduces `totalVestedByToken` by the withdrawn amount (capped at outstanding obligations).

LOW VL1: receive() Does Not Emit Event FIXED

Contract: TreasuryVesting.sol **Function:** receive()

Description: The `receive()` fallback accepts ETH but emits no event. Direct ETH transfers are invisible to off-chain indexers, making it impossible to track vesting funding via events alone.

Impact: Missing audit trail for direct ETH deposits.

Fix: `receive()` now emits `VestingDeposit(ETH_ADDRESS, msg.sender, msg.value)`.

LOW VL2: Unbounded beneficiarySchedules Array ACKNOWLEDGED

Contract: TreasuryVesting.sol **Mapping:** beneficiarySchedules

Description: The `beneficiarySchedules[beneficiary]` array grows without bound as new schedules are created. Functions like `getActiveVestings` and `getTotalVestedForBeneficiary` iterate the entire array. A beneficiary with many schedules could eventually hit gas limits.

Impact: Theoretical gas limit risk for beneficiaries with very many schedules.

Status: Acknowledged — in practice, vesting schedules are created by privileged roles (VESTING_MANAGER) for a limited set of beneficiaries. The iteration cost is acceptable for the expected usage pattern.

TreasuryYield.sol Findings

HIGH YH1: Rebalance Inflates totalDeposited on Deposit Side FIXED

Contract: TreasuryYield.sol **Function:** rebalance

Description: `rebalance` moves capital between strategies but increments `ts.totalDeposited` and `lifetimeDeposited[token]` on the deposit side. Since the tokens were already counted when originally deposited, this double-counts them — inflating global and per-strategy accounting with every rebalance call.

Impact: Accounting corruption — totalDeposited grows unbounded with each rebalance, making withdrawals appear to have negative returns.

Fix: Removed `totalDeposited` and `lifetimeDeposited` increments from the rebalance deposit side, and removed the corresponding decrements from the withdraw side. Rebalance now moves capital without affecting deposit/withdrawal accounting.

MEDIUM YM1: withdrawFromStrategy Holds Tokens Locally Instead of Returning to Treasury FIXED

Contract: TreasuryYield.sol **Function:** withdrawFromStrategy

Description: `withdrawFromStrategy` pulls tokens from the strategy into the TreasuryYield contract but never transfers them back to the treasury. Tokens accumulate in the yield contract rather than being returned to the treasury where they belong.

Impact: Tokens stranded in yield contract; treasury balance does not reflect actual holdings.

Fix: Added `IERC20(token).safeTransfer(treasury, received)` after successful withdrawal from the strategy.

MEDIUM YM2: No Slippage Check on Rebalance Withdraw Side FIXED

Contract: TreasuryYield.sol **Function:** rebalance

Description: The rebalance function withdraws from one strategy and deposits into another, but only checks slippage on the deposit side. The withdrawal from the source strategy has no slippage protection, meaning the source strategy could return significantly fewer tokens than requested without detection.

Impact: Silent value loss if the source strategy returns fewer tokens during rebalance.

Fix: Added slippage check on the withdraw side: `if (slip > MAX_SLIPPAGE_BPS) revert SlippageTooHigh()`.

MEDIUM YM3: emergencyWithdrawFromStrategy Uses Requested Amount for Accounting FIXED

Contract: TreasuryYield.sol **Function:** emergencyWithdrawFromStrategy

Description: `emergencyWithdrawFromStrategy` updates accounting using the requested `amount` parameter rather than the actual tokens received from the strategy. If the strategy returns fewer tokens (due to slippage, fees, or partial withdrawal), the accounting will be overstated — believing more tokens were withdrawn than actually received.

Impact: Accounting desync — `totalDeposited` reduced by more than actually received, corrupting strategy metrics.

Fix: Measures actual received tokens via balance-before/after pattern and uses `received` for all accounting updates.

MEDIUM YM4: Swap-and-Pop Strategy Removal Changes Array Ordering ACKNOWLEDGED

Contract: TreasuryYield.sol **Function:** removeStrategy

Description: `removeStrategy` uses swap-and-pop to remove a strategy from the `tokenStrategies` array. This changes the order of remaining strategies, which could affect any off-chain systems that rely on strategy array indices.

Impact: Off-chain index references become stale after removal.

Status: Acknowledged — swap-and-pop is O(1) and the standard Solidity pattern. On-chain logic uses strategy addresses (not indices), so ordering is irrelevant for contract correctness.

LOW YL1: setTreasury Missing Event Emission FIXED

Contract: TreasuryYield.sol **Function:** setTreasury

Description: `setTreasury` updates the treasury address but emits no event. Off-chain monitoring tools cannot detect when the treasury reference changes, reducing auditability.

Impact: Missing audit trail for treasury address changes.

Fix: Added `TreasuryUpdated(oldTreasury, newTreasury)` event emission.

TreasuryBudgets.sol Findings

HIGH BH1: createBudget Does Not Verify Contract Holds Sufficient Tokens FIXED

Contract: TreasuryBudgets.sol **Function:** createBudget

Description: `createBudget` increments `totalBudgetedByToken` and records a budget without checking whether the contract actually holds enough tokens to cover all outstanding budget obligations. A BUDGET_MANAGER can create budgets far exceeding the contract's balance, and `spend()` will revert later when it tries to transfer.

Impact: Budgets can be over-promised; spending will fail when the contract runs dry.

Fix: Added a post-creation balance check: the contract's token balance must be ≥ `totalBudgetedByToken[token] - totalSpentByToken[token]`. Reverts with `InsufficientBudgetBalance` if underfunded.

HIGH BH2: emergencyWithdraw Does Not Update Budget Accounting FIXED

Contract: TreasuryBudgets.sol **Function:** emergencyWithdraw

Description: `emergencyWithdraw` transfers tokens out but does not reduce `totalBudgetedByToken`. After withdrawal, the accounting claims more tokens are budgeted than the contract holds, causing future `createBudget` balance checks to fail and spends to revert.

Impact: Stale accounting after emergency withdrawal blocks new budgets and breaks spending flows.

Fix: `emergencyWithdraw` now reduces `totalBudgetedByToken` by the withdrawn amount (capped at outstanding obligations).

MEDIUM BM1: depositForBudget Missing Validation and Events FIXED

Contract: TreasuryBudgets.sol **Function:** depositForBudget

Description: `depositForBudget` accepts zero-amount ERC-20 deposits (wasting gas) and emits no event to confirm the deposit was received. Without an event, off-chain indexers cannot track budget funding.

Impact: Wasted gas on zero deposits; no off-chain audit trail for funding.

Fix: Added `if (amount == 0) revert InvalidAmount();` check and a new `BudgetDeposit` event emitted on successful deposit.

MEDIUM BM2: receive() Does Not Emit Event FIXED

Contract: TreasuryBudgets.sol **Function:** receive()

Description: The `receive()` fallback accepts ETH but emits no event. Direct ETH transfers are invisible to off-chain indexers, making it impossible to track budget funding via events alone.

Impact: Missing audit trail for direct ETH deposits.

Fix: `receive()` now emits `BudgetDeposit(ETH_ADDRESS, msg.sender, msg.value)`.

MEDIUM BM3: extendBudgetPeriod Has No Upper Bound FIXED

Contract: TreasuryBudgets.sol **Function:** extendBudgetPeriod

Description: `extendBudgetPeriod` adds `additionalTime` to `endTime` without any upper bound check. An admin could extend a budget indefinitely, and `additionalTime` of 0 is also accepted (no-op with misleading event emission).

Impact: Unbounded extensions; zero-duration extension emits misleading event.

Fix: Added bounds check: `additionalTime` must be > 0 and ≤ `MAX_BUDGET_DURATION`.

MEDIUM BM4: Utilization Alert Fires in Wrong Function FIXED

Contract: TreasuryBudgets.sol **Functions:** allocateToDepartment, spend

Description: The `BudgetUtilizationAlert` event fires inside `allocateToDepartment`, which does not change `budget.spent`. The alert triggers based on spending level during allocation changes, which is misleading. It should fire in `spend()` where spending actually occurs.

Impact: Misleading utilization alerts during allocation, not during actual spending.

Fix: Moved the utilization alert from `allocateToDepartment` to `spend()`.

LOW BL1: Unbounded allBudgetIds Array ACKNOWLEDGED

Contract: TreasuryBudgets.sol **Storage:** allBudgetIds

Description: `allBudgetIds` is append-only. Deactivated budgets are never removed. `getAllBudgetIds()` will eventually hit gas limits if many budgets are created over time.

Impact: Theoretical gas limit risk for view functions iterating all budget IDs.

Status: Acknowledged — budgets are created by privileged roles (BUDGET_MANAGER) and are expected to be limited in number.

LOW BL2: Duplicate getAllBudgets / getAllBudgetIds Functions FIXED

Contract: TreasuryBudgets.sol **Functions:** getAllBudgets, getAllBudgetIds

Description: Both `getAllBudgets()` and `getAllBudgetIds()` return the same `allBudgetIds` array. Duplicated dead code.

Impact: Code duplication and confusing API surface.

Fix: Removed `getAllBudgets()`; kept the more descriptive `getAllBudgetIds()`.

TreasuryMultiSig.sol Findings

HIGH MSH1: proposeChangeThreshold Rejects Unanimity FIXED

Contract: TreasuryMultiSig.sol **Function:** proposeChangeThreshold

Description: The validation `newThreshold >= signerCount` prevents setting the threshold equal to the number of signers (unanimity). With 3 signers, `proposeChangeThreshold(3, ...)` reverts. This should be `newThreshold > signerCount`.

Impact: Cannot configure unanimity requirement; limits governance flexibility.

Fix: Changed `>=` to `>` in the threshold validation, allowing threshold == signerCount.

HIGH MSH2: executeProposal Does Not Re-Validate State for Governance Changes FIXED

Contract: TreasuryMultiSig.sol **Function:** executeProposal

Description: Between proposal creation and execution (up to 7-day expiry), state may have changed. `AddSigner` does not re-check that the signer doesn't already exist or that `signerCount < MAX_SIGNERS`. `RemoveSigner` does not re-check that the signer still exists or that `signerCount > MIN_SIGNERS`. `ChangeThreshold` does not re-validate against current `signerCount`.

Impact: Stale proposals can corrupt signer tracking or set invalid thresholds.

Fix: Added execution-time re-validation for all three governance proposal types: duplicate/count checks for AddSigner, existence/minimum/threshold checks for RemoveSigner, and bounds validation for ChangeThreshold.

MEDIUM MSM1: revokeRole Unconditionally Decrements signerCount FIXED

Contract: TreasuryMultiSig.sol **Function:** revokeRole

Description: `revokeRole` decrements `signerCount` whenever `role == SIGNER_ROLE`, even if the account did not actually have the role (the parent `revokeRole` is a no-op in that case). Repeated calls can underflow `signerCount`, corrupting signer tracking.

Impact: `signerCount` desync; potential underflow corruption.

Fix: Checks `hasRole(SIGNER_ROLE, account)` before the parent call; only decrements and removes from list if the account actually held the role.

MEDIUM MSM2: deposit and receive() Missing Events FIXED

Contract: TreasuryMultiSig.sol **Functions:** deposit, receive()

Description: `deposit()` accepts tokens and `receive()` accepts ETH, but neither emits any event. Incoming funds are invisible to off-chain monitoring tools.

Impact: No audit trail for deposits into the multi-sig contract.

Fix: Both functions now emit a `Deposit` event. Added zero-amount check to `deposit()`.

MEDIUM MSM3: emergencyWithdraw Does Not Update Metrics FIXED

Contract: TreasuryMultiSig.sol **Function:** emergencyWithdraw

Description: `emergencyWithdraw` transfers tokens but does not update `totalWithdrawn[token]`, breaking withdrawal metrics. Normal proposal-based transfers do update this metric.

Impact: `totalWithdrawn` understates actual withdrawals after emergency use.

Fix: Added `totalWithdrawn[token] += amount;` before the transfer.

MEDIUM MSM4: Signer Removal Can Break Threshold Invariant FIXED

Contract: TreasuryMultiSig.sol **Function:** executeProposal (RemoveSigner)

Description: `proposeRemoveSigner` checks `signerCount > MIN_SIGNERS` but does not check whether removal would leave `signerCount < requiredSignatures`. With 3 signers and threshold 3, removing one leaves 2 signers but threshold stays 3 — making all future proposals unexecutable.

Impact: Governance deadlock if signer removal drops count below threshold.

Fix: Covered by MSH2 — execution-time re-validation now checks `signerCount - 1 ≥ requiredSignatures` before removing a signer.

LOW MSL1: Linear Scan in _removePendingProposal ACKNOWLEDGED

Contract: TreasuryMultiSig.sol **Function:** _removePendingProposal

Description: `_removePendingProposal` does a linear scan of the `pendingProposals` array (up to `MAX_PENDING_PROPOSALS = 50`). Gas cost grows linearly with the number of pending proposals.

Impact: Minor gas inefficiency; bounded by MAX_PENDING_PROPOSALS (50).

Status: Acknowledged — the array is bounded at 50 entries, keeping gas costs within acceptable limits. A mapping-based index would add complexity for minimal gain.

TreasuryNFTFees.sol Findings

HIGH NFH1: distributeFees Rounding Dust Loss FIXED

Contract: TreasuryNFTFees.sol **Function:** distributeFees

Description: `distributeFees` calculates each recipient's share as `(total * bps) / 10_000`. Integer division truncates, and the per-recipient rounding errors accumulate. When the sum of all cuts is less than the total, the difference remains stuck in the contract with no mechanism to recover it.

Impact: Rounding dust (up to `recipients.length - 1` wei per distribution) permanently locks in the contract, compounding over many distributions.

Fix: The last recipient now receives `total - distributed` (the remainder) instead of a truncated BPS calculation, eliminating dust accumulation.

HIGH NFH2: accumulateFees Accepts Zero Amount FIXED

Contract: TreasuryNFTFees.sol **Function:** accumulateFees

Description: `accumulateFees` had no check for `amount == 0`. A caller could repeatedly invoke it with zero, emitting misleading `FeesAccumulated` events and inflating event counts without any actual fee transfer.

Impact: Misleading accounting events and wasted gas. Off-chain indexers counting events would report inflated fee activity.

Fix: Added `if (amount == 0) revert InvalidAmount();` at the top of `accumulateFees`.

MEDIUM NFM1: emergencyWithdraw Does Not Update Fee Accounting FIXED

Contract: TreasuryNFTFees.sol **Function:** emergencyWithdraw

Description: `emergencyWithdraw` transfers tokens out but does not reduce `accumulatedFeesByToken` or `_accrued`. After withdrawal, these mappings claim more fees exist than the contract holds, causing future `distributeFees` calls to attempt transfers exceeding the actual balance.

Impact: Stale accounting after emergency withdrawal breaks fee distribution and creates phantom balances.

Fix: `emergencyWithdraw` now reduces both `accumulatedFeesByToken` and `_accrued` by the withdrawn amount (capped at tracked totals).

MEDIUM NFM2: Dual Fee Accounting Paths ACKNOWLEDGED

Contract: TreasuryNFTFees.sol **Function:** collectNFTFee / accumulateFees

Description: The contract has two independent fee collection paths: `collectNFTFee` → `withdrawFees` (per-collection fees withdrawn by a recipient) and `accumulateFees` → `distributeFees` (accumulated fees split among BPS-weighted recipients). These paths use separate accounting mappings and do not interact, which could confuse integrators.

Impact: Potential confusion; no direct exploit since the paths track separate balances independently.

Status: Acknowledged — the dual paths serve distinct use cases (per-collection vs aggregated distribution) and are intentionally separate.

MEDIUM NFM3: receive() Does Not Emit Event FIXED

Contract: TreasuryNFTFees.sol **Function:** receive()

Description: The `receive()` function accepts ETH without emitting an event. Direct ETH transfers are invisible to off-chain indexers that rely on event logs.

Impact: ETH deposits via direct transfer are not tracked off-chain, making accounting and reconciliation difficult.

Fix: `receive()` now emits `EthReceived(msg.sender, msg.value)`.

MEDIUM NFM4: setTreasury Missing Event FIXED

Contract: TreasuryNFTFees.sol **Function:** setTreasury

Description: `setTreasury` updates the treasury address without emitting an event. Treasury address changes are high-impact governance actions that should be auditable on-chain.

Impact: Treasury migrations are invisible to off-chain monitoring; malicious changes go undetected.

Fix: `setTreasury` now emits `TreasuryUpdated(oldTreasury, newTreasury)`.

LOW NFL1: Misleading Balance Pre-Check in distributeFees FIXED

Contract: TreasuryNFTFees.sol **Function:** distributeFees

Description: `distributeFees` contained a balance pre-check comparing `IERC20(token).balanceOf(address(this))` against the distribution total. This check is redundant since `safeTransfer` already reverts on insufficient balance, and the pre-check creates a false sense of security for non-standard tokens.

Impact: No functional impact; the misleading check adds gas and could mask the real failure point.

Fix: Removed the redundant balance pre-check. The `safeTransfer` call provides the authoritative revert on insufficient balance.

LOW NFL2: Unused accumulatedFees State Variable ACKNOWLEDGED

Contract: TreasuryNFTFees.sol **Function:** (state variable)

Description: The `accumulatedFees` state variable (a single uint256) is never written to internally — the contract uses `accumulatedFeesByToken` (a mapping) for actual accounting. The variable exists solely to satisfy the `INFTFeeManager` interface requirement.

Impact: The variable always returns 0, which could confuse callers expecting aggregate fee data.

Status: Acknowledged — removing the variable would break the `INFTFeeManager` interface. It is retained with a comment documenting the interface constraint.

TreasuryNFTManager.sol Findings

MEDIUM NMM1: setNftVault Missing Zero-Address Check FIXED

Contract: TreasuryNFTManager.sol **Function:** setNftVault

Description: `setNftVault` accepted `address(0)` as a valid vault address, which would disable all NFT operations since every function checks `nftVault != address(0)`.

Impact: An admin could accidentally brick all NFT deposit/withdrawal operations by setting the vault to the zero address.

Fix: Added `if (vault == address(0)) revert ZeroAddress();` to `setNftVault`.

MEDIUM NMM2: Constructor Missing Input Validation FIXED

Contract: TreasuryNFTManager.sol **Function:** constructor

Description: The constructor accepted zero addresses for both `_treasury` and `_nftVault` parameters. Since `treasury` is immutable, a zero address would permanently break all treasury interactions with no way to fix it.

Impact: Deployment with zero addresses creates a permanently broken contract that must be redeployed.

Fix: Added `if (_treasury == address(0)) revert ZeroAddress();` and `if (_nftVault == address(0)) revert ZeroAddress();` to the constructor.

LOW NML1: Unbounded Batch Operations FIXED

Contract: TreasuryNFTManager.sol **Function:** batchDepositNFT / batchWithdrawNFT

Description: `batchDepositNFT` and `batchWithdrawNFT` had no upper bound on array length. A caller could pass arrays large enough to exceed the block gas limit, causing the transaction to revert.

Impact: Denial-of-service risk via gas exhaustion on oversized batches.

Fix: Added `MAX_BATCH_SIZE = 50` constant and `if (nfts.length > MAX_BATCH_SIZE) revert BatchTooLarge();` to both batch functions.

IntegratorHub.sol Findings

HIGH IHH1: receive() Uses String Revert Instead of Custom Error FIXED

Contract: IntegratorHub.sol **Function:** receive()

Description: The `receive()` function used `revert("No direct ETH")` which costs more gas than a custom error and is inconsistent with the rest of the contract which uses custom errors throughout (e.g., `UnexpectedETH`).

Impact: Wastes gas on every accidental ETH send. The string-based revert is also inconsistent with the contract's error pattern, making ABI parsing harder for off-chain tools.

Fix: Replaced `revert("No direct ETH")` with `revert UnexpectedETH()`.

HIGH IHH2: batchConsume Unbounded Array Length FIXED

Contract: IntegratorHub.sol **Function:** batchConsume

Description: `batchConsume` iterates over `routers.length` with no upper bound. A treasury caller could pass arrays large enough to exhaust the block gas limit.

Impact: Gas exhaustion DoS on batch consume operations.

Fix: Added `MAX_BATCH_SIZE = 50` constant and `if (routers.length > MAX_BATCH_SIZE) revert BatchTooLarge();`.

MEDIUM IHM1: routerTokens Array Grows Without Bound ACKNOWLEDGED

Contract: IntegratorHub.sol **Function:** accrue

Description: Every new token accrued for a router is pushed to `routerTokens[router]`. Even after consumption zeroes the balance, the token entry remains. Over time this array grows unbounded, and `totalAccrued()` must iterate the entire array including stale zero-balance entries.

Impact: Gas cost of `totalAccrued()` increases monotonically. Eventually the view function may exceed gas limits for routers that have traded many tokens.

Status: Acknowledged — the array is append-only by design and `totalAccrued()` already filters zero balances. Adding cleanup logic would increase `accrue()` gas cost.

MEDIUM IHM2: _consumeAccrued Sets lastSweep for Inactive Apps FIXED

Contract: IntegratorHub.sol **Function:** _consumeAccrued

Description: When an app is inactive/unpaid, `_consumeAccrued` still updated `a.lastSweep` to `block.timestamp`. If the app was later reactivated, the sweep timer reset from the last consume call rather than from reactivation, potentially skipping an overdue payout period.

Impact: Reactivated apps lose their accrued sweep eligibility; `isDue()` returns false immediately after reactivation even if the app was due before deactivation.

Fix: `_consumeAccrued` no longer updates `lastSweep` for inactive apps — only active apps get their sweep timer updated.

MEDIUM IHM3: minAccrualAmount Set Twice in Constructor FIXED

Contract: IntegratorHub.sol **Function:** constructor

Description: `minAccrualAmount` was initialized to `1e6` at the storage declaration, then immediately overwritten to `0` in the constructor. The storage-level default was dead code and misleading.

Impact: No functional impact — the constructor value wins. But the contradictory initialization confuses auditors and future developers.

Fix: Removed the storage-level initialization (`= 1e6`), keeping only the constructor assignment as the single source of truth.

MEDIUM IHM4: reactivateMyApp Does Not Reset lastSweep FIXED

Contract: IntegratorHub.sol **Function:** reactivateMyApp

Description: When a router reactivated its app, `lastSweep` retained its value from before deactivation. If the app was inactive for a long time, `isDue()` would immediately return `true` even though no new fees had accrued during the inactive period, triggering premature consume cycles.

Impact: Premature sweep trigger on reactivation, creating unnecessary gas-wasting consume calls that return (0,0).

Fix: `reactivateMyApp` now resets `lastSweep = uint64(block.timestamp)`.

LOW **IHL1: getBatchAccrued Unbounded Array** FIXED

Contract: IntegratorHub.sol **Function:** getBatchAccrued

Description: `getBatchAccrued` had no max batch size. While it's a view function, extremely large arrays could still fail due to gas limits on `eth_call`.

Impact: Minor — view function only, but inconsistent with the bounded `batchConsume`.

Fix: Added the same `MAX_BATCH_SIZE` check for consistency.

NFTVault.sol Findings

HIGH **NVH1: emergencyNFTTransfer Does Not Update _holdings** FIXED

Contract: NFTVault.sol **Function:** emergencyNFTTransfer

Description: `emergencyNFTTransfer` transferred an NFT out of the vault but did not remove the `tokenId` from `_holdings[nft]`. After the transfer, `getNFTsByContract(nft)` still reported the token as held, and `getNFTBalance` for ERC-721 tokens returned 1 even though the vault no longer owned it.

Impact: Stale holdings data after emergency transfer. Any code relying on `getNFTsByContract` or `getNFTBalance` sees phantom tokens. Subsequent `withdrawTo` calls for the token would revert since the vault doesn't own it.

Fix: `emergencyNFTTransfer` now removes the tokenId from `_holdings[nft]` after the transfer (for ERC-1155, only when balance reaches 0).

HIGH **NVH2: emergencyNFTTransfer Missing Zero-Address Check** FIXED

Contract: NFTVault.sol **Function:** emergencyNFTTransfer

Description: The `to` parameter was not validated against `address(0)`. For ERC-721, `safeTransferFrom` to `address(0)` reverts (OpenZeppelin checks), but for ERC-1155 the behavior depends on the implementation — some may allow it, burning the token permanently.

Impact: Accidental token burn if admin passes zero address for an ERC-1155 token.

Fix: Added `if (to == address(0)) revert InvalidAddress();`.

MEDIUM NVM1: setTokenAllowed Can Strand NFTs ACKNOWLEDGED

Contract: NFTVault.sol **Function:** setTokenAllowed

Description: An owner can set `allowedToken[nft] = false` while the vault still holds tokens from that NFT contract. Since `withdrawTo` checks `allowedToken`, those tokens become locked — only `emergencyNFTTransfer` (which requires `locked = true`) can recover them.

Impact: NFTs stranded in the vault if allowlist is updated carelessly. Emergency recovery is possible but requires locking the entire vault.

Status: Acknowledged — this is a policy decision. The emergency transfer path provides recovery. The risk is documented.

MEDIUM NVM2: Batch Operations Unbounded Array Length FIXED

Contract: NFTVault.sol **Function:** batchDepositFrom / batchWithdrawTo

Description: Neither `batchDepositFrom` nor `batchWithdrawTo` had an upper bound on array length. Each iteration performs `supportsInterface` external calls (2 per item), which are expensive. Large arrays can exceed the block gas limit.

Impact: Gas exhaustion DoS on batch operations.

Fix: Added `MAX_BATCH_SIZE = 50` constant and check to both batch functions.

LOW NVL1: Repeated supportsInterface Calls in Batch Operations ACKNOWLEDGED

Contract: NFTVault.sol **Function:** batchDepositFrom / batchWithdrawTo

Description: Each iteration in the batch loop calls `supportsInterface` twice (ERC-721 and ERC-1155 checks) via external calls. If the same NFT contract appears multiple times in the batch, the same interface checks are repeated, wasting gas.

Impact: Gas inefficiency when batching multiple tokens from the same contract. Minor since the `onlyTreasury` caller controls the input.

Status: Acknowledged — caching would add complexity for a gas optimization that benefits only repeated-contract batches.

LOW NVL2: getNFTBalance ERC-721 Check Uses _holdings Instead of ownerOf FIXED

Contract: NFTVault.sol **Function:** getNFTBalance

Description: For ERC-721 tokens, `getNFTBalance` checks `_holdings[nft].contains(tokenId)` instead of calling `IERC721(nft).ownerOf(tokenId)`. If `_holdings` gets out of sync (e.g., after `emergencyNFTTransfer`), this returns stale data.

Impact: Returns incorrect balance if holdings are stale. Resolved by NVH1 fix which ensures emergency transfers update `_holdings`.

Fix: Resolved by NVH1 — `emergencyNFTTransfer` now keeps `_holdings` in sync, preventing the desync that caused stale data.

Donation.sol Findings

HIGH DH1: donateByCause Iterates All Organizations Without Bound FIXED

Contract: Donation.sol **Function:** donateByCause / _countVerifiedOrgsForCause

Description: `donateByCause` loops through ALL organizations twice — once in `_countVerifiedOrgsForCause` and once in the main distribution loop. There was no upper bound on `organizationCount`. As organizations accumulate, gas cost grows linearly and eventually exceeds the block gas limit.

Impact: Permanent DoS — once enough organizations exist, `donateByCause` becomes uncallable.

Fix: Added `MAX_ORGANIZATIONS = 500` constant enforced in `addOrganization`, bounding the iteration cost.

MEDIUM DM1: fundsRaised Mixes Token Denominations ACKNOWLEDGED

Contract: Donation.sol **Function:** donate / donateETH / donateByCause / donateFromWinnings

Description: `fundsRaised` is a single `uint256` that sums donations across all tokens (USDC with 6 decimals, WETH with 18 decimals, native ETH). The total is meaningless when mixing different decimal scales.

Impact: Misleading accounting metric. Off-chain tools reading `fundsRaised` cannot derive meaningful totals.

Status: Acknowledged — the metric is informational and not used in any on-chain logic. A per-token tracking mapping would add storage cost for a cosmetic improvement.

MEDIUM DM2: donateETH Forwards ETH via call to Arbitrary Wallet ACKNOWLEDGED

Contract: Donation.sol **Function:** donateETH

Description: `donateETH` sends ETH via `org.wallet.call{value: msg.value}("")`. If the wallet is a contract with a malicious `receive()` function, it could revert to grief donors or consume excessive gas.

Impact: A malicious or buggy wallet contract can block ETH donations to that organization. The reentrancy guard mitigates re-entrancy, but griefing remains possible.

Status: Acknowledged — the wallet is set by a trusted admin role. Adding EOA-only checks would prevent legitimate multi-sig wallets from receiving donations.

MEDIUM DM3: donate Does Not Validate tokenAddress FIXED

Contract: Donation.sol **Function:** donate / donateByCause / donateFromWinnings

Description: `donate` called `IERC20(tokenAddress).safeTransferFrom(...)` without validating that `tokenAddress` is non-zero or a legitimate contract. Passing `address(0)` results in unclear low-level revert messages.

Impact: Poor UX — unclear revert messages for invalid token addresses.

Fix: Added `if (tokenAddress == address(0)) revert InvalidTokenAddress();` to all three donation functions.

LOW DL1: Unnecessary unchecked Increment FIXED

Contract: Donation.sol **Function:** addOrganization

Description: `organizationCount++` was wrapped in `unchecked`. The gas savings are immaterial relative to the function's multiple storage writes and string operations.

Impact: Negligible gas savings with reduced safety clarity.

Fix: Removed the `unchecked` wrapper.

LOW DL2: verifyOrganization Allows Redundant Unverify FIXED

Contract: Donation.sol **Function:** verifyOrganization

Description: `verifyOrganization` checked for duplicate verification but not duplicate unverification. Calling `verifyOrganization(id, false)` on an already-unverified org emitted redundant events.

Impact: Redundant events cluttering off-chain logs.

Fix: Added symmetry check: `if (!verified && !org.verified) revert OrganizationNotVerified();`

PredictionAMM.sol Findings

HIGH PAH1: redeem Pro-Rata Rounding Dust Loss FIXED

Contract: PredictionAMM.sol **Function:** redeem

Description: The pro-rata payout `(shares * pool.collateral) / total` truncates on each redemption. As `totalWinningShares` and `pool.collateral` decrease with each redeemer, rounding compounds. The last redeemer receives less than their fair share, and dust remains stranded in the pool.

Impact: Collateral dust permanently locked after all winners redeem. Winners collectively receive less than the full collateral.

Fix: When `shares ≥ totalWinningShares` (last redeemer), the payout is set to all remaining `pool.collateral`, eliminating dust.

HIGH PAH2: seedLiquidity Missing Market Status Check FIXED

Contract: PredictionAMM.sol **Function:** seedLiquidity

Description: `seedLiquidity` checked that the market exists (`mv.creator != address(0)`) but did not verify `mv.status == MarketStatus.Active`. A pool could be seeded for a Resolved, Voided, or Disputed market, locking collateral with no way to trade.

Impact: Collateral locked in a pool for a non-active market. Provider must wait for resolution to call `removeLiquidity`.

Fix: Added `if (mv.status != MarketStatus.Active) revert PoolNotActive();` after the existence check.

HIGH PAH3: Constructor Does Not Validate _predictionMarket FIXED

Contract: PredictionAMM.sol **Function:** constructor

Description: The constructor accepted `_predictionMarket` without validating it's a non-zero address. Since `predictionMarket` is immutable, a zero address permanently breaks all functions that call `predictionMarket.getMarket()`.

Impact: Deployment with zero address creates a permanently broken contract.

Fix: Added `if (_predictionMarket == address(0)) revert ZeroAddress();`.

MEDIUM PAM1: sell Can Underflow pool.collateral FIXED

Contract: PredictionAMM.sol **Function:** sell

Description: `sell` computed `payout` via LMSR math and then did `pool.collateral -= payout` without checking that payout does not exceed the available collateral. An underflow would panic-revert with an unclear error.

Impact: Unclear panic error instead of a descriptive revert when LMSR payout exceeds collateral.

Fix: Added `if (payout > pool.collateral) revert InsufficientLiquidity();` before the subtraction.

MEDIUM PAM2: removeLiquidity Can Drain Collateral Before All Redeemers Claim FIXED

Contract: PredictionAMM.sol **Function:** removeLiquidity

Description: After a market is Resolved, the provider could call `removeLiquidity` and drain ALL remaining collateral, including collateral owed to winning share holders who hadn't redeemed yet.

Impact: Winners who haven't redeemed yet lose their payout. The provider gets collateral that should go to winning share holders.

Fix: Added check: `if (winningSharesCounted[marketId] && totalWinningShares[marketId] > 0) revert WinningSharesPending();`

MEDIUM PAM3: redeem Allows Claiming on ResolutionPending Markets FIXED

Contract: PredictionAMM.sol **Function:** redeem

Description: `redeem` accepted both `MarketStatus.Resolved` and `MarketStatus.ResolutionPending`. During `ResolutionPending`, the winning outcome may not be finalized — it could change if a dispute is filed.

Impact: Premature redemption on potentially incorrect outcome. If the outcome changes via dispute, the redeemer keeps tokens they shouldn't have.

Fix: Changed to only accept `MarketStatus.Resolved`.

LOW PAL1: Uniform Price Rounding in getOutcomePrices FIXED

Contract: PredictionAMM.sol **Function:** getOutcomePrices

Description: When the pool is not active, `getOutcomePrices` returns uniform prices as `MAX_PRICE / n`. For non-divisible outcome counts (e.g., n=3), prices don't sum to `MAX_PRICE`.

Impact: Cosmetic — prices don't sum to `MAX_PRICE` for non-divisible outcome counts.

Fix: Added rounding remainder to the last outcome, matching the pattern used in `_lmsrPrices`.

PredictionMarket.sol Findings

CRITICAL PMC1: ecrecover Returns address(0) on Invalid Signatures FIXED

Contract: PredictionMarket.sol **Function:** resolveDispute

Description: `ecrecover` returns `address(0)` for malformed signatures. In the M-of-N dispute council verification path, if any `disputeCouncilMembers[j]` were `address(0)`, an attacker could forge "valid" signatures. The legacy single-council path had the same risk if `disputeCouncil` were ever set to `address(0)` (partially mitigated by the check at the top, but not in the recovery comparison itself).

Impact: Signature bypass — anyone could resolve disputes with forged signatures if any council member address is zero.

Fix: Added `if (recovered == address(0)) continue;` after each `ecrecover` call in both M-of-N and legacy paths.

CRITICAL PMC2: Duplicate Signature Counting in M-of-N Verification FIXED

Contract: PredictionMarket.sol **Function:** resolveDispute

Description: The M-of-N dispute council verification iterated through signatures and incremented `validCount` each time a match was found, but did not track which council members had already been counted. A single council member's signature submitted multiple times in the `disputeCouncilSigs` array would increment `validCount` each time, allowing a 1-of-N attack to pass an M-of-N quorum.

Impact: Complete bypass of multi-sig dispute council quorum — a single council member could unilaterally resolve disputes.

Fix: Added a `usedMask` bitmap that tracks which council member indices have been matched, preventing duplicate counting.

HIGH PMH1: withdrawFees Uses require String Instead of Custom Error FIXED

Contract: PredictionMarket.sol **Function:** withdrawFees

Description: `require(ok, "ETH transfer failed")` is inconsistent with the rest of the contract which uses custom errors. String reverts waste gas on deployment (stored in bytecode) and at revert time (ABI-encoded).

Impact: Gas waste and inconsistent error handling pattern.

Fix: Replaced with `if (!ok) revert EthTransferFailed();`.

HIGH PMH2: resolveDispute ETH Transfers Use require Strings FIXED

Contract: PredictionMarket.sol **Function:** resolveDispute

Description: Both bond distribution paths (disputer refund and feeRecipient transfer) used `require(ok, "ETH transfer failed")` instead of custom errors.

Impact: Same as PMH1 — gas waste and inconsistency.

Fix: Both paths now use `if (!ok) revert EthTransferFailed();`.

HIGH PMH3: setAMM Allows Setting to address(0) FIXED

Contract: PredictionMarket.sol **Function:** setAMM

Description: No zero-address check on `setAMM`. Setting AMM to `address(0)` silently disables AMM functionality, inconsistent with `setFeeRecipient` and `setDisputeCouncil` which both validate non-zero.

Impact: Silent AMM disablement; markets created after would have no market maker.

Fix: Added `if (_amm == address(0)) revert ZeroAddress();`.

HIGH PMH4: setDisputeCouncilMultisig Accepts Zero-Address and Duplicate Members FIXED

Contract: PredictionMarket.sol **Function:** setDisputeCouncilMultisig

Description: The members array was not validated for zero addresses or duplicates. A zero-address member combined with PMC1 would allow anyone to forge valid signatures. Duplicate members reduce effective quorum (e.g., 3-of-5 with 2 duplicates becomes effectively 3-of-4).

Impact: Weakened dispute council security through zero-address or duplicate member acceptance.

Fix: Added validation loop: each member is checked for `address(0)` and uniqueness against all preceding members.

HIGH PMH5: createMarket Does Not Validate Token Address FIXED

Contract: PredictionMarket.sol **Function:** createMarket

Description: `params.token` is stored without validation. If `address(0)`, markets created without initial liquidity would have a broken settlement token, and any later AMM seeding or trading attempts would fail with confusing errors.

Impact: Markets created with invalid token address; downstream failures with poor error messages.

Fix: Added `if (params.token == address(0)) revert ZeroAddress();`.

MEDIUM PMM1: disputeWindow Can Be Set to Zero ACK

Contract: PredictionMarket.sol **Function:** setDisputeWindow

Description: No minimum bound on `setDisputeWindow`. Setting to 0 makes `finalizeResolution` callable in the same block as resolution, effectively eliminating the dispute protection period.

Impact: Dispute protection bypassed if owner sets window to 0.

Status: Acknowledged — admin-only function; considered a design decision for operational flexibility.

MEDIUM PMM2: creationFee and disputeBond Can Be Set to Zero ACK

Contract: PredictionMarket.sol **Functions:** setCreationFee, setDisputeBond

Description: No minimum enforcement on creation fee or dispute bond. Setting either to 0 removes anti-spam protection (creation fee) or skin-in-the-game incentive (dispute bond).

Impact: Spam market creation or frivolous disputes without economic cost.

Status: Acknowledged — admin-only functions; zero values may be intentional during testing or promotional periods.

MEDIUM PMM3: Dispute Bond Refund Can Block Resolution Permanently FIXED

Contract: PredictionMarket.sol **Function:** resolveDispute

Description: If `m.disputer` is a contract that rejects ETH (no `receive` or `fallback`), the bond refund `.call{value: bondToReturn}` fails, causing `resolveDispute` to revert. This permanently blocks dispute resolution for the market.

Impact: Market permanently stuck in Disputed state if disputer cannot receive ETH.

Fix: Added fallback: if disputer transfer fails, bond is sent to `feeRecipient` instead. If both fail, reverts with `EthTransferFailed`.

MEDIUM PMM4: Bare receive() Accepts Arbitrary ETH Deposits FIXED

Contract: PredictionMarket.sol **Function:** receive

Description: The bare `receive() external payable {}` allowed anyone to send ETH to the contract with no tracking. This ETH gets mixed with creation fees and dispute bonds, inflating `withdrawFees` payouts. The contract only needs to receive ETH through its `payable` functions (`createMarket`, `disputeResolution`).

Impact: Untracked ETH deposits inflate fee withdrawals.

Fix: Removed `receive()` — contract only accepts ETH through its payable functions.

LOW PML1: getMarketsByCategory Unbounded Return FIXED

Contract: PredictionMarket.sol **Function:** getMarketsByCategory

Description: The `limit` parameter is user-controlled with no cap. A very large limit could cause out-of-gas on view calls.

Impact: View call failures for large categories.

Fix: Added `if (limit > MAX_BATCH_SIZE) limit = MAX_BATCH_SIZE;` with `MAX_BATCH_SIZE = 100`.

LOW PML2: getCreatorMarkets Returns Unbounded Array FIXED

Contract: PredictionMarket.sol **Function:** getCreatorMarkets

Description: Returns the entire `creatorMarkets[creator]` array with no pagination. A prolific creator could cause view call failures.

Impact: View call failures for prolific creators.

Fix: Added offset/limit pagination matching `getMarketsByCategory` pattern, with `MAX_BATCH_SIZE` cap.

LOW PML3: Missing whenNotPaused on Resolution and Dispute Functions FIXED

Contract: PredictionMarket.sol **Functions:** resolveMarket, disputeResolution, resolveDispute

Description: These lifecycle functions did not check pause state. If the contract is paused (emergency), resolution and disputes could still proceed, undermining the pause mechanism.

Impact: Emergency pause does not fully halt protocol operations.

Fix: Added `whenNotPaused` modifier to all three functions.

---

20. Conclusion

This deep audit identified **125 findings** across 20 SBET Protocol contracts — 21 in the core betting contracts (SBET.sol and related modules), 15 in the treasury system (SBETTreasury.sol), 10 in the fee manager (TreasuryFeeManager.sol), 7 in the vesting contract (TreasuryVesting.sol), 6 in the yield contract (TreasuryYield.sol), 8 in the budget manager (TreasuryBudgets.sol), 7 in the multi-sig governance (TreasuryMultiSig.sol), 8 in the NFT fee manager (TreasuryNFTFees.sol), 3 in the NFT manager (TreasuryNFTManager.sol), 7 in the integrator hub (IntegratorHub.sol), 6 in the NFT vault (NFTVault.sol), 6 in the donation manager (Donation.sol), 7 in the prediction AMM (PredictionAMM.sol), and 14 in the prediction market lifecycle (PredictionMarket.sol). Of these, **105 have been fixed** and **20 acknowledged** as acceptable design decisions or compatibility constraints.

SBET.sol Critical Fixes

C1: EIP-712 domain separator spacing corrected for standard library interop
C2: Oracle price truncation now validated with bounds checks before uint32 cast
C3: Claim path rerouted through _settlePosition to enforce grader fees
C4/C5: NFT finalization now sets matchFinalPrice and supports multi-batch processing
C6: matchOrders nonce marking moved outside the per-right-order loop

SBETTreasury.sol Critical & High Fixes

TC1/TC2: Deposit paths (receive + batchDeposit) now enforce pause and lock guards
TH1: Daily withdrawal limit arithmetic made underflow-safe
TH2: Zero-amount deposits rejected in batchDeposit
TH3: Payout address verification now applies universally (EOA + contracts)

TreasuryFeeManager.sol High Fixes

FMH1: Timelocked operations now re-validate all constraints at execution time
FMH2: Zero-amount fee accumulation rejected

TreasuryVesting.sol High Fix

VH1: createVesting now verifies contract holds sufficient tokens for all outstanding obligations

TreasuryYield.sol High Fix

YH1: Rebalance no longer inflates totalDeposited — capital moves without double-counting

TreasuryBudgets.sol High Fixes

BH1: createBudget now verifies contract holds sufficient tokens for all outstanding obligations
BH2: emergencyWithdraw reduces totalBudgetedByToken to keep accounting consistent

TreasuryMultiSig.sol High Fixes

MSH1: proposeChangeThreshold now allows unanimity (threshold == signerCount)
MSH2: executeProposal re-validates governance changes at execution time

TreasuryNFTFees.sol High Fixes

NFH1: distributeFees last recipient gets remainder to eliminate rounding dust
NFH2: accumulateFees rejects zero-amount accumulation

IntegratorHub.sol High Fixes

IHH1: receive() uses custom error UnexpectedETH instead of string revert
IHH2: batchConsume bounded by MAX_BATCH_SIZE (50)

NFTVault.sol High Fixes

NVH1: emergencyNFTTransfer now updates _holdings after transfer
NVH2: emergencyNFTTransfer validates to != address(0)

Donation.sol High Fix

DH1: addOrganization enforces MAX_ORGANIZATIONS (500) to bound donateByCause loops

PredictionAMM.sol High Fixes

PAH1: Last redeemer gets all remaining collateral (eliminates rounding dust)
PAH2: seedLiquidity checks market status is Active
PAH3: Constructor validates _predictionMarket != address(0)

PredictionMarket.sol Critical & High Fixes

PMC1/PMC2: ecrecover address(0) bypass prevented; bitmap prevents duplicate signature counting in M-of-N dispute council
PMH1/PMH2: All ETH transfer error handling converted to custom errors
PMH3: setAMM validates non-zero address
PMH4: Dispute council members validated for zero-address and uniqueness
PMH5: createMarket validates settlement token address

All 16 critical and 34 high-severity findings have been fixed across both audit rounds. The 23 acknowledged items (M1, M4, M8, L1, TL3, FML1, FML3, VL2, YM4, BL1, MSL1, NFM2, NFL2, IHM1, NVM1, NVL1, DM1, DM2, PMM1, PMM2, M-01f, M-04f, M-08f) carry no direct exploit risk and are retained for architectural compatibility or as documented design decisions.

---

21. Follow-Up Audit — Round 2

Date: 2026-02-25 · **Scope:** Deep review of SBET.sol and all inherited modules (SBETStorage, SBETTrading, SBETPool, SBETNFT, SBETClaim, SBETMath, SBETQuery, ISBETInterfaces)

Commits: `a1dd98b`, `df8f45f`

Round 2 Findings Summary

SEVERITY	COUNT	FIXED	ACKNOWLEDGED
CRITICAL	6	6	0
HIGH	4	4	0
MEDIUM	8	5	3
LOW	3	3	0
Total	**21**	**18**	**3**

CRITICAL C-01f: EIP-712 Domain Separator Spacing Mismatch FIXED

Contract: SBET.sol **Line:** 107

Description: The EIP-712 domain type string contained spurious spaces after commas (`"uint256 chainId, address verifyingContract, bytes32 salt"`), deviating from the canonical EIP-712 specification. Standard libraries produce the hash without spaces, causing signature verification failures for standard-compliant signers.

Impact: Frontends using canonical EIP-712 spacing would produce mismatched domain hashes, breaking all signature verification.

Fix: Removed spurious spaces: `"uint256 chainId,address verifyingContract,bytes32 salt"`.

CRITICAL C-02f: Oracle Price Truncation and Missing Bounds Checks FIXED

Contract: SBET.sol **Function:** executeFinalization() **Lines:** 193–204

Description: `uint32(uint256(price))` silently truncated Chainlink `int256` answers (8–18 decimals) to 32 bits. No checks for `price ≤ 0` or `price ≥ MAX_PRICE` existed before the cast.

Impact: Match finalized at incorrect/garbage price; irreversible fund loss.

Fix: Added `if (price ≤ 0) revert BadFinalPrice()` and `if (uint256(price) ≥ MAX_PRICE) revert BadFinalPrice()` before the `uint32` cast.

CRITICAL C-03f: Standalone `claim()` Bypassed Grader Fees FIXED

Contract: SBETClaim.sol **Function:** claim() **Lines:** 39–66

Description: The base virtual `claim()` computed payoff and transferred directly without routing through `_settlePosition()`, the only path that deducts grader fees. Also contained an unreachable `safeTransferFrom` branch for negative payoffs.

Impact: Users could bypass grader fees entirely; dead code implied losers needed to cooperate for winners to claim.

Fix: Rerouted `claim()` and `batchClaimPositions()` through `_settlePosition()`. Removed dead negative-payoff branch.

CRITICAL C-04f: NFT Finalization Missing `matchFinalPrice` FIXED

Contract: SBETNFT.sol **Function:** _executeFinalizeMatchNFTs() **Lines:** 279–306

Description: `_executeFinalizeMatchNFTs` set `matchFinalized[matchId] = true` but never set `matchFinalPrice`. P2P positions on the same matchId became unclaimable (`_computePayoff` returned 0 for longs when `matchFinalPrice == 0`).

Impact: All P2P longs on NFT-finalized matches lost their entire position value.

Fix: Added `matchFinalPrice` assignment based on outcome: SideA → `MAX_PRICE - 1`, SideB → `1`, None → `MAX_PRICE / 2`.

CRITICAL C-05f: Batch NFT Finalization Broken After First Call FIXED

Contract: SBETNFT.sol **Function:** _executeFinalizeMatchNFTsBatch() **Lines:** 308–328

Description: Every call set `matchFinalized[matchId] = true` unconditionally, then subsequent batch calls reverted on `if (matchFinalized[matchId]) revert MatchIsFinalized()`. Only the first batch ever ran; remaining NFTs were permanently locked.

Impact: Permanent NFT lockup for any match requiring multiple finalization batches.

Fix: First call sets `matchFinalized` and `matchFinalPrice`. Subsequent calls verify `matchWinningSide[matchId] == outcome` to prevent tampering, and skip the finalization step.

CRITICAL C-06f: `matchOrders` Left Nonce Marked Per Right Order FIXED

Contract: SBETTrading.sol **Function:** _matchSingleRight() **Lines:** 1014–1018

Description: `_markNonceUsed(left.maker, left.nonce)` was called inside the loop for each right order match. After the first right order filled, subsequent iterations operated on stale position snapshots with the nonce already marked.

Impact: Double-counting of left order fills; position accounting corruption.

Fix: Moved `_markNonceUsed(left.maker, left.nonce)` to `_executeMatchOrders()` after the loop, called once when `totalLeftFilled > 0`.

HIGH H-01f: Grader Fee Distribution Order-Dependent FIXED

Contract: SBET.sol **Function:** claimGraderFees() **Lines:** 446–451

Description: The “last grader gets remainder” logic depended on claim order, not grader identity. Race conditions between graders determined who received the rounding dust bonus.

Impact: Non-deterministic fee allocation; unfair rounding advantage.

Fix: All graders now receive the same floor-divided share (`pool / numReporters`). Rounding dust stays in the pool.

HIGH H-02f: `joinPoolBet` Validates `poolId` After Storage Dereference FIXED

Contract: SBETPool.sol **Function:** joinPoolBet(), _executeJoinPoolBet() **Lines:** 48–67

Description: `Pool storage p = pools[poolId]` was dereferenced before the bounds check. For non-existent pools, `p.numOutcomes` was 0, so `outcome ≥ p.numOutcomes` fired `InvalidOutcome` instead of the correct `InvalidPoolId`.

Impact: Misleading error for invalid pool IDs; confusing UX and integration bugs.

Fix: Moved `poolId` bounds check before storage access in both functions.

HIGH H-03f: `_finalizeMatch` Allows `finalPrice == 0` FIXED

Contract: SBET.sol **Function:** _finalizeMatch() **Lines:** 474–476

Description: The check was `if (finalPrice > MAX_PRICE)`, not `≥ MAX_PRICE`. A `finalPrice` of 0 passed validation, giving longs a payoff of 0 and shorts their full position. Combined with C-02f, exploitable via oracle price truncation.

Impact: Forced zero payoff for all longs; complete loss of position value.

Fix: Changed to `if (finalPrice == 0 || finalPrice ≥ MAX_PRICE) revert BadFinalPrice()`.

HIGH H-04f: Dead `safeTransferFrom` on Negative Payoff FIXED

Contract: SBETClaim.sol **Function:** _settlePosition() **Lines:** 149–151

Description: `_computePayoff` always returns ≥ 0 (uses absolute magnitude). The `else if (payoff < 0)` branch calling `safeTransferFrom` was unreachable dead code that implied losers needed to cooperate for winners to claim.

Impact: Dead code creating false security assumptions about settlement flow.

Fix: Removed dead branch. Added comment explaining payoff is always non-negative.

MEDIUM M-01f: Hardcoded SALT in EIP-712 Domain ACKNOWLEDGED

Contract: SBET.sol **Line:** 71

Description: Fixed salt adds no entropy beyond `chainId + verifyingContract`.

Impact: Cosmetic — no direct exploit. The salt is part of the committed EIP-712 domain and changing it would break existing signatures.

Fix: Acknowledged — retained for backward compatibility.

MEDIUM M-02f: `notEmergency` Modifier Auto-Clears Emergency State FIXED

Contract: SBET.sol **Lines:** 131–135

Description: The modifier silently set `emergencyPaused = false` as a side effect when any user called a guarded function after `emergencyEndTime`. State changes in modifiers are surprising, and this created a race window where the owner couldn't extend an emergency.

Impact: Emergency could be silently cleared by any user; owner loses ability to extend pause window.

Fix: Modifier now reverts for both active and expired-but-uncleared emergencies. Added explicit `clearEmergency()` owner-only function with `EmergencyCleared` event.

MEDIUM M-03f: `priceDivide` Uses `assert()` Instead of Custom Error FIXED

Contract: SBETMath.sol **Lines:** 34–35

Description: `assert()` consumes all remaining gas on failure. `effectiveBalance` had the same issue.

Impact: Full gas consumption on revert; poor UX and wasted gas.

Fix: Replaced with `revert InvalidPriceDivideArgs()` and `revert InvalidEffectiveBalancePrice()`.

MEDIUM M-04f: Dead Code — Base Virtual Functions Duplicate `_execute*` Internals ACKNOWLEDGED

Contract: SBETTrading.sol **Lines:** 86–132

Description: `trade()`, `matchOrders()`, `cancelAll()`, etc. have complete implementations in the base abstract contract, duplicating the `_execute*` internal variants. The SBET.sol overrides call `_execute*`, making the base implementations dead code.

Impact: Maintenance risk — dead code may drift from actual behavior.

Fix: Acknowledged — the virtual/override pattern is the intended inheritance design. Base functions serve as the interface contract.

MEDIUM M-05f: Pool Claim Doesn't Reset `userStake` FIXED

Contract: SBETPool.sol **Line:** 137

Description: Normal settlement only cleared `userOutcomeStake[msg.sender][p.outcome]` but not `userStake[msg.sender]`, causing `getUserActivePools` to return stale data.

Impact: Stale pool membership data; incorrect frontend state.

Fix: Added `p.userStake[msg.sender] = 0` in both claim paths.

MEDIUM M-06f: Unchecked `tradeSize` Addition FIXED

Contract: SBETTrading.sol **Lines:** 261–263

Description: `tradeSize = t.longAmount + t.shortAmount` was in an `unchecked` block with a manual overflow check after. The pattern was fragile and the comment was misleading.

Impact: Overflow risk in a critical path; unclear intent.

Fix: Replaced with checked arithmetic. Solidity 0.8.x handles overflow natively.

MEDIUM M-07f: Dead If-Block in `finalizeWithGraders` FIXED

Contract: SBET.sol **Lines:** 352–354

Description: `if (matchRecoveryDeadline[matchId] == 0) { /* no-op */ }` — empty block with only a comment.

Impact: Dead code; misleading intent.

Fix: Removed.

MEDIUM M-08f: Broken ETH Trading Path ACKNOWLEDGED

Contract: SBETTrading.sol **Lines:** 534–539

Description: `_applyTradeAll` routes to `_applyETHBalances` when `o.token == address(0)`, but `_adjustETH` reverts on negative deltas. The ETH path is broken for any trade where the short side owes.

Impact: Dead code — upstream checks ensure the ETH path is never reached.

Fix: Acknowledged — guarded by upstream validation that validates `token != address(0)` and `treasury.isAllowedToken(token)`.

LOW L-01f: Duplicate Constants Between `SBETMath` and `SBETStorage` FIXED

Contract: SBETMath.sol, SBETStorage.sol

Description: `MAX_PRICE` and `MAX_POSITION_MAGNITUDE` independently defined in both files. Library `internal` constants can't be accessed cross-contract, so true deduplication is impossible.

Impact: Maintenance risk — constants may drift out of sync.

Fix: Added prominent sync comments in both files with `IMPORTANT` annotation.

LOW L-02f: `SBETQuery.getPoolPayoutSimulation` Overflow Risk FIXED

Contract: SBETQuery.sol **Line:** 286

Description: `(totalStaked * userStake) / outcomeTotal` risks overflow on large pool values. The main contract correctly uses `Math.mulDiv`.

Impact: Incorrect query results for large pools.

Fix: Replaced with `Math.mulDiv(totalStaked, userStake, outcomeTotal)` for 512-bit intermediate math.

LOW L-03f: `getPositionValue` Missing Price Bounds Check FIXED

Contract: SBET.sol **Lines:** 792–805

Description: No validation on `currentPrice` input. If `currentPrice > MAX_PRICE`, the short path underflows (`MAX_PRICE - uint256(currentPrice)`). No early return for zero positions.

Impact: Potential underflow in view function; incorrect position valuation.

Fix: Added `if (currentPrice == 0 || currentPrice ≥ MAX_PRICE_U32) revert InvalidPriceRange(currentPrice)` and `if (position == 0) return 0`.

Round 2 Conclusion

All 6 critical and 4 high-severity follow-up findings have been fixed. The 3 acknowledged items (M-01f, M-04f, M-08f) carry no direct exploit risk and are retained for backward compatibility or as documented design decisions. Combined with the initial audit, the protocol has undergone **146 total findings** across 2 audit rounds — all remediated or acknowledged with documented rationale.

22. Glossary

TERM	DEFINITION
Access Control	Mechanisms that restrict which addresses or roles can invoke specific contract functions, typically implemented via modifiers like `onlyOwner` or role-based systems (e.g., OpenZeppelin AccessControl).
Acknowledged	A finding status indicating the development team has reviewed the issue and accepts the current behavior as a deliberate design decision or acceptable trade-off, with documented rationale.
Bond	Collateral deposited by a market creator or dispute participant that is forfeited if their claim is found to be invalid, incentivizing honest behavior.
Collateral	Tokens locked in a smart contract to back a position, bet, or obligation. Collateral is returned or redistributed based on the outcome.
Cross-Contract Interaction	A call from one smart contract to another, which introduces trust boundaries and potential reentrancy or state inconsistency risks.
Dead Code	Code paths that are unreachable during normal execution. While not directly exploitable, dead code increases attack surface and audit complexity.
Domain Separator	A structured hash defined by EIP-712 that uniquely identifies a signing domain (contract address, chain ID, name, version) to prevent signature replay across contexts.
Dust	Tiny residual token amounts (typically <1 wei of value) left over from integer division in fee distribution or reward calculations.
Emergency Withdrawal	A privileged function allowing protocol administrators to extract funds from a contract during critical failures, typically bypassing normal withdrawal limits and time locks.
Finalization	The process of resolving a match or market outcome on-chain, after which positions can be settled and payouts distributed.
Finding	A discrete security observation identified during the audit, classified by severity (Critical, High, Medium, Low) and status (Fixed, Acknowledged).
Fuzzing	An automated testing technique that feeds random or semi-random inputs to a program to discover crashes, assertion failures, and invariant violations.
Gas Griefing	An attack where a malicious actor causes a transaction to consume excessive gas, potentially making it revert or become economically impractical.
Grader	An authorized address responsible for reporting match outcomes to the SBET contract, receiving a share of settlement fees for this service.
Invariant	A property that must always hold true throughout contract execution (e.g., total deposits ≥ total withdrawals). Invariant violations indicate bugs.
LMSR	Logarithmic Market Scoring Rule — an automated market maker algorithm used for prediction markets that provides bounded loss for the liquidity provider and dynamic pricing based on demand.
Nonce	A unique number used once per signed order to prevent replay attacks. Once an order nonce is consumed, the same signed message cannot be reused.
Oracle	An external data source that provides off-chain information (e.g., sports scores, token prices) to on-chain smart contracts.

TERM	DEFINITION
Reentrancy	A vulnerability where an external call allows the called contract to re-enter the calling function before its state updates are complete, potentially draining funds.
Remediated	A finding status indicating the issue has been fully addressed through a code change that eliminates the vulnerability or concern.
Severity	The assessed impact level of a finding: **Critical** (direct fund loss), **High** (significant impact), **Medium** (moderate risk), **Low** (minor/cosmetic).
Slippage	The difference between the expected price/odds of a trade and the actual execution price, often caused by pool imbalance changes between submission and execution.
Timelock	A delay mechanism requiring a minimum waiting period between proposing and executing a sensitive operation, giving stakeholders time to review or veto.

23. Acronyms

ACRONYM	EXPANSION
ABI	Application Binary Interface — the encoding specification for calling contract functions and decoding return data.
AMM	Automated Market Maker — a smart contract that provides liquidity and pricing algorithmically instead of using an order book.
CCIP	Cross-Chain Interoperability Protocol — Chainlink's standard for secure cross-chain messaging and token transfers.
CEI	Checks-Effects-Interactions — a Solidity design pattern that prevents reentrancy by performing all state changes before external calls.
DAO	Decentralized Autonomous Organization — a governance structure where decisions are made by token-holder voting rather than a central authority.
DeFi	Decentralized Finance — financial services implemented as permissionless smart contracts on public blockchains.
EIP	Ethereum Improvement Proposal — a design document proposing new features, standards, or processes for the Ethereum ecosystem.
EIP-712	A standard for typed structured data hashing and signing, enabling human-readable signature requests in wallets.
EOA	Externally Owned Account — an Ethereum account controlled by a private key (as opposed to a contract account).
ERC-20	The standard interface for fungible tokens on Ethereum, defining transfer, approval, and balance query functions.
ERC-721	The standard interface for non-fungible tokens (NFTs) on Ethereum, defining ownership, transfer, and metadata functions.
ERC-1155	A multi-token standard supporting both fungible and non-fungible tokens in a single contract with batch operations.
EVM	Ethereum Virtual Machine — the runtime environment that executes smart contract bytecode on Ethereum and compatible chains.
FOK	Fill-Or-Kill — an order execution policy requiring the entire order to be filled immediately or cancelled entirely.
GTC	Good-Til-Cancelled — an order that remains active until explicitly cancelled by the user or filled by a counterparty.
IOC	Immediate-Or-Cancel — an order that fills as much as possible immediately, then cancels any unfilled remainder.
LMSR	Logarithmic Market Scoring Rule — a cost-function-based automated market maker for prediction markets.
NFT	Non-Fungible Token — a unique blockchain token (typically ERC-721 or ERC-1155) representing ownership of a distinct asset.
P2P	Peer-to-Peer — direct interaction between two participants without intermediaries, as in SBET's signed-order matching system.
RBAC	Role-Based Access Control — a permission model where function access is granted based on assigned roles rather than individual addresses.
RPC	Remote Procedure Call — the protocol used to communicate with Ethereum nodes for reading state and submitting transactions.
SBET	Sports Betting Token — the native token of the SBET Protocol used for staking, governance, and fee distribution.
SDK	Software Development Kit — a packaged set of tools, libraries, and documentation for building on a platform.

ACRONYM	EXPANSION
TVL	Total Value Locked — the aggregate value of assets deposited in a DeFi protocol's smart contracts.

24. Disclaimer

Notice: This audit report is provided by Versus Security for the exclusive use of the SBET Protocol team. It represents findings at the time of the audit and does not constitute a guarantee of security. Smart contracts remain subject to risks including but not limited to undiscovered vulnerabilities, third-party dependencies, and evolving attack vectors. This report should not be used as the sole basis for investment decisions. Versus Security accepts no liability for losses arising from use of or reliance on this report.

Versus Security · versus-sec.com

Cyber Operations · Digital Warfare · Cyber Defense