Predict.Fun Protocol

Smart Contract Security Audit Report

Audited by: TenArmor

Audit dates: Dec 15, 2025 - Dec 26, 2025

Content

1. Introduction

1.1 About TenArmor

TenArmor is a leading Web3 security firm dedicated to safeguarding blockchain ecosystems. We offer cutting-edge security solutions tailored to the complexities of decentralized technologies. With our flagship products, TenMonitor for real-time threat detection and response and TenTrace for a one-stop AML solution, we provide comprehensive protection against evolving threats. Our expertise spans smart contract auditing, cryptocurrency tracing, and beyond, making TenArmor a trusted partner for organizations aiming to secure their digital assets in the decentralized world.

Learn more about us:
Official Website: TenArmor.com
X: @TenArmor, @TenArmorAlert
E-mail: team@tenarmor.com
TenMonitor: TenMonitor.com
TenTrace: TenTrace.com

1.2 Disclaimer

This report represents an analysis performed within a defined scope and time frame, based on the materials and documentation provided. It is not exhaustive and does not encompass all potential vulnerabilities.

A smart contract security review aims to identify as many vulnerabilities as possible within the given constraints of time, resources, and expertise. However, it cannot guarantee the complete absence of vulnerabilities or the complete security of the project. This report does not serve as an endorsement of any specific project or team.

To enhance security, we strongly recommend conducting subsequent security reviews, implementing bug bounty programs, and maintaining ongoing on-chain monitoring.

1.3 Risk Classification

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

2. Executive Summary

2.1 About Predict.Fun Protocol

Predict.Fun implements a centralized-operator prediction market built on Gnosis Conditional Tokens (ERC1155) for binary YES/NO outcome positions, with an RFQ/order-book style exchange (CTFExchange) that fills and matches off-chain signed orders on-chain. Market resolution is driven by a UMA-compatible Optimistic Oracle stack (UmaCompatibleOptimisticOracle), and fees are collected/refunded via fee modules (FeeModuleV2 / NegRiskFeeModuleV2) with an auxiliary handler to withdraw and merge accumulated position-token fees into collateral. The system additionally supports "NegRisk" multi-question markets through an adapter/operator layer that enforces at-most-one-true semantics and enables position conversion for capital efficiency, and an optional yield-bearing mode that deposits collateral into Venus either directly in a yield-bearing CTF implementation or via a yield-bearing wrapped collateral used by the NegRisk adapter.

2.2 Audit Scope

Repository: https://github.com/PredictDotFun/prediction-market/tree/main/contracts

Audit Commit Hash: 164fe622d225627cb3d7bbb7085770f74df64825

2.3 Findings Count

Severity	Count
Critical	0
High	0
Medium	0
Low	6
Informational	6
Total Findings	**12**

3. Findings Summary

ID	Description	Status
[L-1]	Unexpected-balance sweep in _matchOrders can produce unintended fund distribution	Acknowledged
[L-2]	Orders with takerAmount == 0 can lead to “maker gives, receives 0”	Acknowledged
[L-3]	getExpectedPayouts can return incorrect payouts if request isn’t settled	Acknowledged
[L-4]	questionCount overflow can corrupt packed MarketData state	Acknowledged
[L-5]	Missing zero-address checks in yield-bearing configuration and transfers	Acknowledged
[L-6]	_disableUnderlying may permanently fail when underlying cannot cover recorded principal	Acknowledged
[I-1]	outcomeSlotCount==256 allowed in prepareCondition but breaks fullIndexSet computation	Acknowledged
[I-2]	Use SafeERC20 consistently for transfer/transferFrom	Acknowledged
[I-3]	Use forceApprove (or safe approve pattern) for ERC20 approvals	Acknowledged
[I-4]	Add explicit array-length checks to avoid out-of-bounds reverts	Acknowledged
[I-5]	Several comments do not match implemented access control or parameter meaning	Acknowledged
[I-6]	vToken verification uses `isVToken()` only; compromised `YIELD_MANAGER_ROLE` can cause fund loss	Acknowledged

Feedback Details: https://docs.google.com/spreadsheets/d/1tw1jepLoO2BasoMgZZAv8XiHv1h81XeX8bHFMGWgAEo

4. Findings

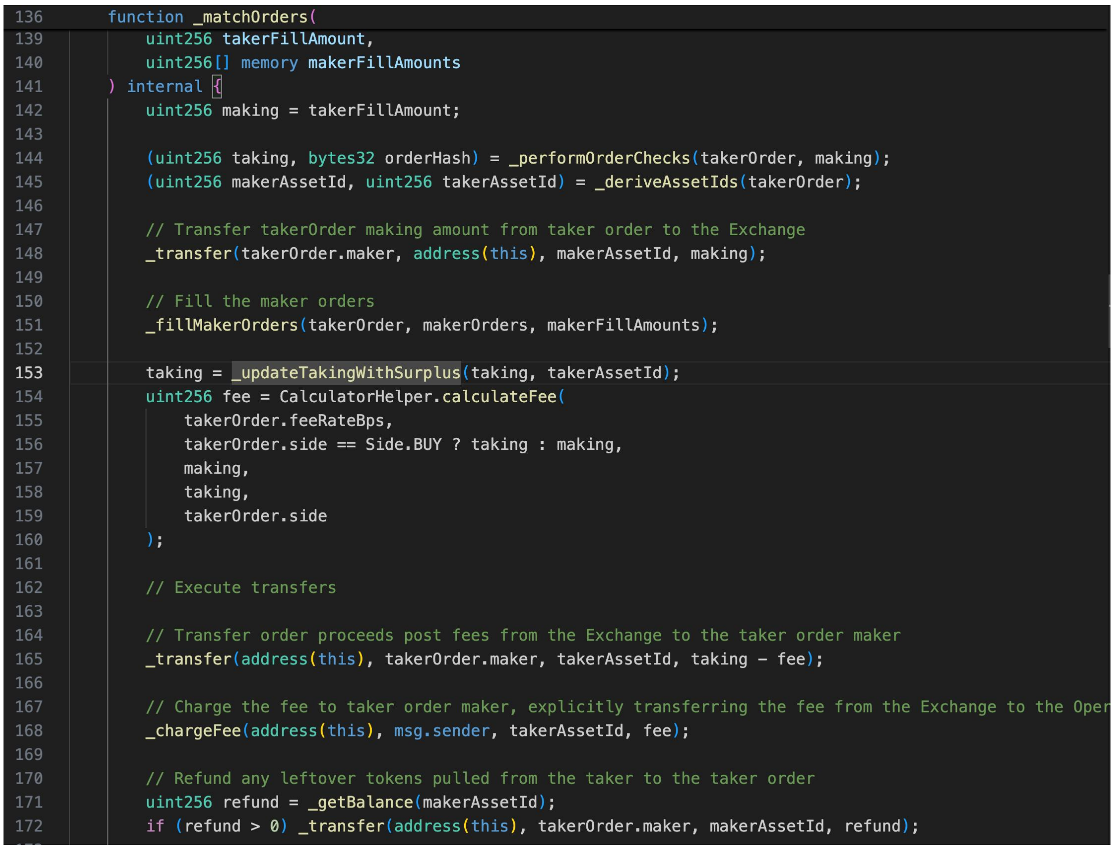
4.1 Low Risk

[L-1] Unexpected-balance sweep in _matchOrders can produce unintended fund distribution

Description:

In Trading._matchOrders, the taker-side taking amount is updated to the Exchange's full current balance of the taker asset via _updateTakingWithSurplus, and any leftover maker-asset balance is refunded to takerOrder.maker. This means if the Exchange contract is accidentally sent extra collateral/CTF tokens, the function will treat that as part of the execution and sweep it primarily to takerOrder.maker, with the operator also receiving a fee on the enlarged taking.

Exchange/mixins/Trading.sol Line#153 and Line#171



```
136     function _matchOrders(
139         uint256 takerFillAmount,
140         uint256[] memory makerFillAmounts
141     ) internal {
142         uint256 making = takerFillAmount;
143
144         (uint256 taking, bytes32 orderHash) = _performOrderChecks(takerOrder, making);
145         (uint256 makerAssetId, uint256 takerAssetId) = _deriveAssetIds(takerOrder);
146
147         // Transfer takerOrder making amount from taker order to the Exchange
148         _transfer(takerOrder.maker, address(this), makerAssetId, making);
149
150         // Fill the maker orders
151         _fillMakerOrders(takerOrder, makerOrders, makerFillAmounts);
152
153         taking = _updateTakingWithSurplus(taking, takerAssetId);
154         uint256 fee = CalculatorHelper.calculateFee(
155             takerOrder.feeRateBps,
156             takerOrder.side == Side.BUY ? taking : making,
157             making,
158             taking,
159             takerOrder.side
160         );
161
162         // Execute transfers
163
164         // Transfer order proceeds post fees from the Exchange to the taker order maker
165         _transfer(address(this), takerOrder.maker, takerAssetId, taking - fee);
166
167         // Charge the fee to taker order maker, explicitly transferring the fee from the Exchange to the Oper
168         _chargeFee(address(this), msg.sender, takerAssetId, fee);
169
170         // Refund any leftover tokens pulled from the taker to the taker order
171         uint256 refund = _getBalance(makerAssetId);
172         if (refund > 0) _transfer(address(this), takerOrder.maker, makerAssetId, refund);
```

Impact:

It creates confusing and potentially unfair outcomes and complicates accounting and refund correctness assumptions across the fee stack.

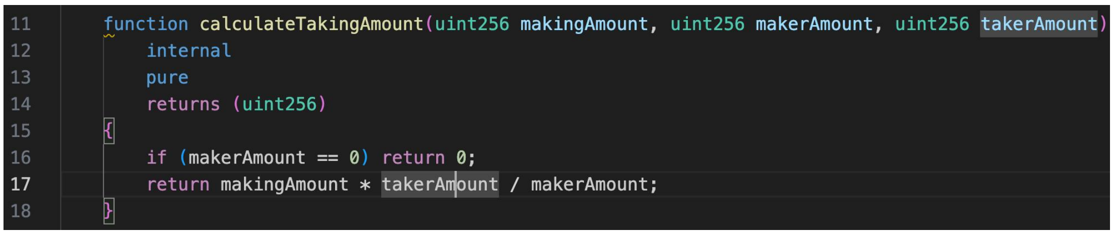
Recommendation:

Compute taking and refund based only on amounts attributable to the current execution (strict accounting), and handle unexpected balances via an explicit sweep() (admin/treasury) function that can periodically recover stray tokens.

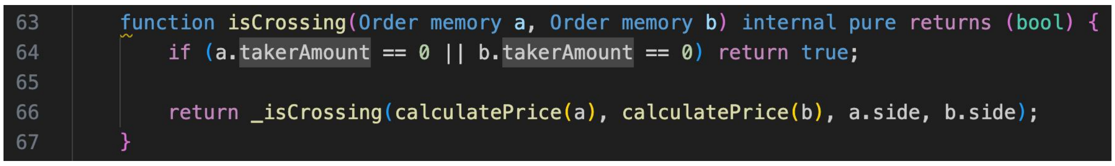
[L-2] Orders with takerAmount == 0 can lead to “maker gives, receives 0”

Description:
Order validation does not forbid order.takerAmount == 0. takingAmount is computed by CalculatorHelper.calculateTakingAmount(making, makerAmount, takerAmount), so with takerAmount == 0 the computed taking can be zero, enabling fills where the maker transfers making but receives taking - fee (potentially 0). This is possible in both _fillOrder and _matchOrders paths.

Exchange/libraries/CalculatorHelper.sol Line#17 and Line#64



```
11      function calculateTakingAmount(uint256 makingAmount, uint256 makerAmount, uint256 takerAmount)
12          internal
13          pure
14          returns (uint256)
15      {
16          if (makerAmount == 0) return 0;
17          return makingAmount * takerAmount / makerAmount;
18      }
```



```
63      function isCrossing(Order memory a, Order memory b) internal pure returns (bool) {
64          if (a.takerAmount == 0 || b.takerAmount == 0) return true;
65
66          return _isCrossing(calculatePrice(a), calculatePrice(b), a.side, b.side);
67      }
```

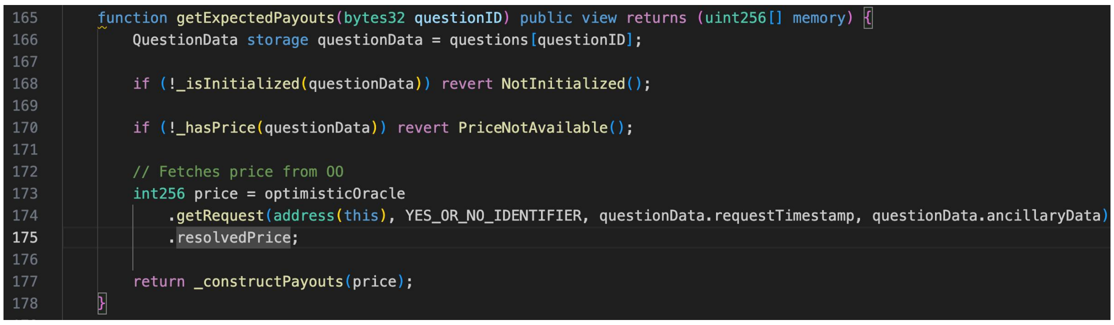
Impact:
It increases risk of user misconfiguration, unintended donation orders, and operational incidents if off-chain validation fails.

Recommendation:
Add a contract-level invariant require(order.takerAmount > 0) (or an explicit error) in _validateOrder, and also enforce this in off-chain order ingestion.

[L-3] getExpectedPayouts can return incorrect payouts if request isn’t settled

Description:
getExpectedPayouts checks _hasPrice(questionData) and then reads optimisticOracl e.getRequest(...).resolvedPrice directly. In oracle implementations where hasPrice == true can become true before resolvedPrice is written (i.e., request is “eligible” but not yet settled), this can return the default zero value and thus an incorrect payou t vector.

UMA/UmaCompatibleCtfAdapter.sol Line#175



```
165     function getExpectedPayouts(bytes32 questionID) public view returns (uint256[] memory) {
166         QuestionData storage questionData = questions[questionID];
167 
168         if (!_isInitialized(questionData)) revert NotInitialized();
169 
170         if (!_hasPrice(questionData)) revert PriceNotAvailable();
171 
172         // Fetches price from OO
173         int256 price = optimisticOracle
174             .getRequest(address(this), YES_OR_NO_IDENTIFIER, questionData.requestTimestamp, questionData.ancillaryData)
175             .resolvedPrice;
176 
177         return _constructPayouts(price);
178     }
```

Impact:

View function, but can mislead UIs/integrations and cause incorrect expected payout displays.

Recommendation:

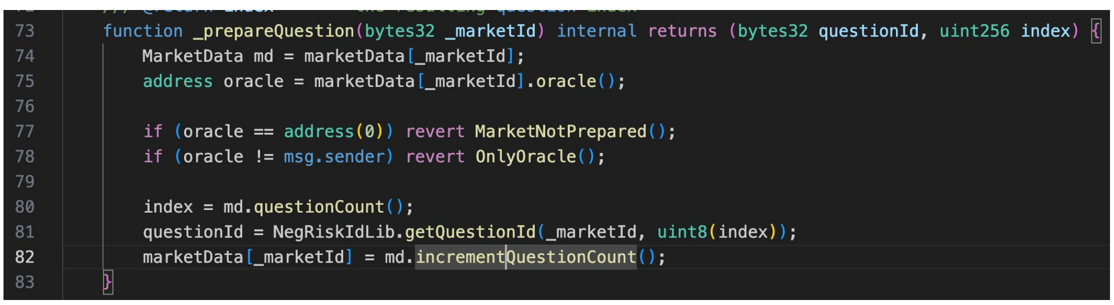
Gate getExpectedPayouts on a settled state (or a request-state field) rather than hasPrice alone; alternatively compute expected payouts from proposed state when not settled, or document that this value is only reliable after resolve().

[L-4] questionCount overflow can corrupt packed MarketData state

Description:

MarketDataLib.incrementQuestionCount explicitly does not check whether questionCount is already at max. MarketStateManager._prepareQuestion uses index = md.questionCount() then writes marketData[_marketId] = md.incrementQuestionCount() with no upper bound. Since MarketData is a packed bytes32, overflowing the lowest byte can carry into other fields (e.g., determined/result/fee/oracle bits depending on carry), corrupting market state.

NegRisk/modules/MarketDataManager.sol
Line#82



```
73      function _prepareQuestion(bytes32 _marketId) internal returns (bytes32 questionId, uint256 index) {
74          MarketData md = marketData[_marketId];
75          address oracle = marketData[_marketId].oracle();
76 
77          if (oracle == address(0)) revert MarketNotPrepared();
78          if (oracle != msg.sender) revert OnlyOracle();
79 
80          index = md.questionCount();
81          questionId = NegRiskIdLib.getQuestionId(_marketId, uint8(index));
82          marketData[_marketId] = md.incrementQuestionCount();
83      }
```

Impact:

Low severity in normal usage but is a latent state-corruption risk and can break markets if reached or abused.

Recommendation:

Add an explicit cap (e.g., revert when index == type(uint8).max) in _prepareQuestion (or in the library) and document max questions per market.

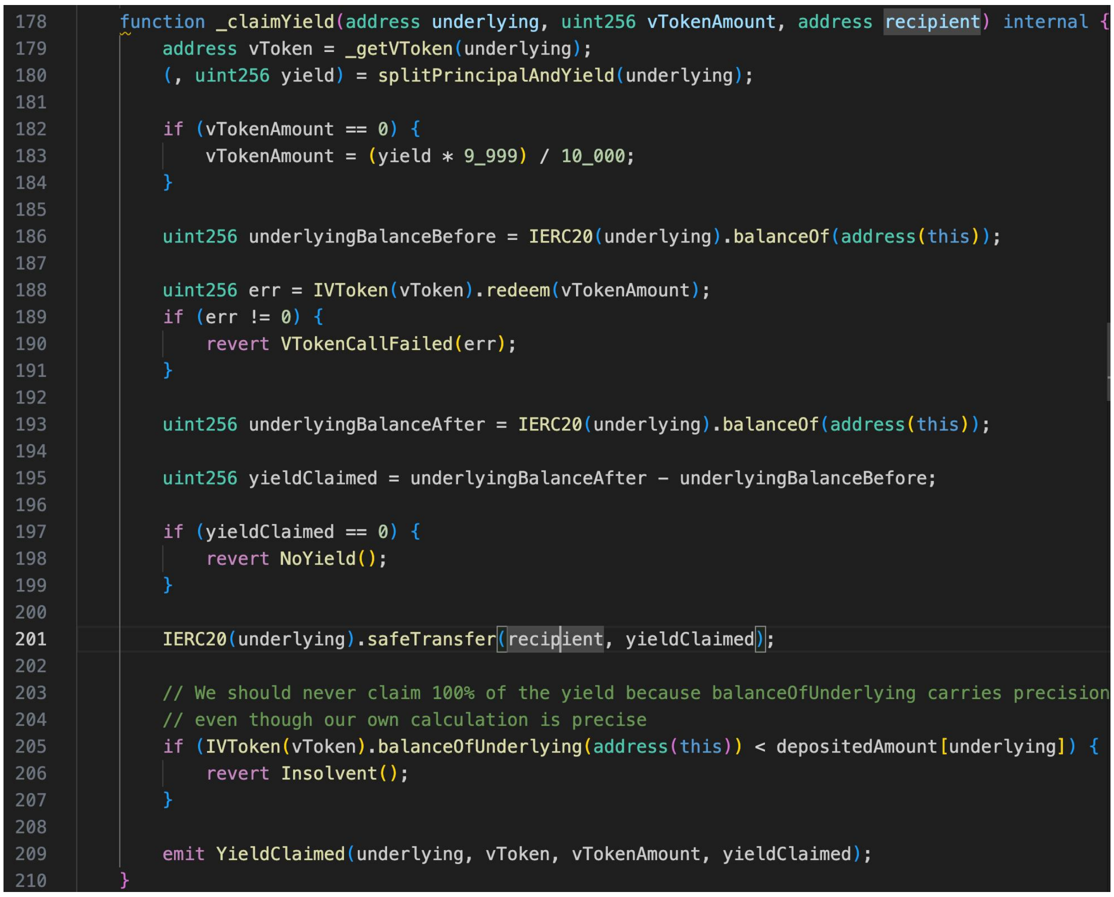
[L-5] Missing zero-address checks in yield-bearing configuration and transfers

Description:

- Venus._claimYield transfers yield to recipient without checking recipient != address(0).
- YieldBearingWrappedCollateral.updateYieldManager allows setting yieldManager to address(0) without validation.

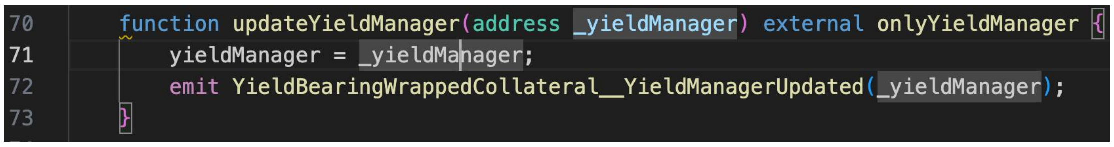
Both are role-gated but represent avoidable operational footguns.

YieldBearing/Venus.sol Line#201



```
178     function _claimYield(address underlying, uint256 vTokenAmount, address recipient) internal {
179         address vToken = _getVToken(underlying);
180         (, uint256 yield) = splitPrincipalAndYield(underlying);
181 
182         if (vTokenAmount == 0) {
183             vTokenAmount = (yield * 9_999) / 10_000;
184         }
185 
186         uint256 underlyingBalanceBefore = IERC20(underlying).balanceOf(address(this));
187 
188         uint256 err = IVToken(vToken).redeem(vTokenAmount);
189         if (err != 0) {
190             revert VTokenCallFailed(err);
191         }
192 
193         uint256 underlyingBalanceAfter = IERC20(underlying).balanceOf(address(this));
194 
195         uint256 yieldClaimed = underlyingBalanceAfter - underlyingBalanceBefore;
196 
197         if (yieldClaimed == 0) {
198             revert NoYield();
199         }
200 
201         IERC20(underlying).safeTransfer(recipient, yieldClaimed);
202 
203         // We should never claim 100% of the yield because balanceOfUnderlying carries precision
204         // even though our own calculation is precise
205         if (IVToken(vToken).balanceOfUnderlying(address(this)) < depositedAmount[underlying]) {
206             revert Insolvent();
207         }
208 
209         emit YieldClaimed(underlying, vToken, vTokenAmount, yieldClaimed);
210     }
```

YieldBearing/YieldBearingWrappedCollateral.sol Line#71



```
70     function updateYieldManager(address _yieldManager) external onlyYieldManager {
71         yieldManager = _yieldManager;
72         emit YieldBearingWrappedCollateral__YieldManagerUpdated(_yieldManager);
73     }
```

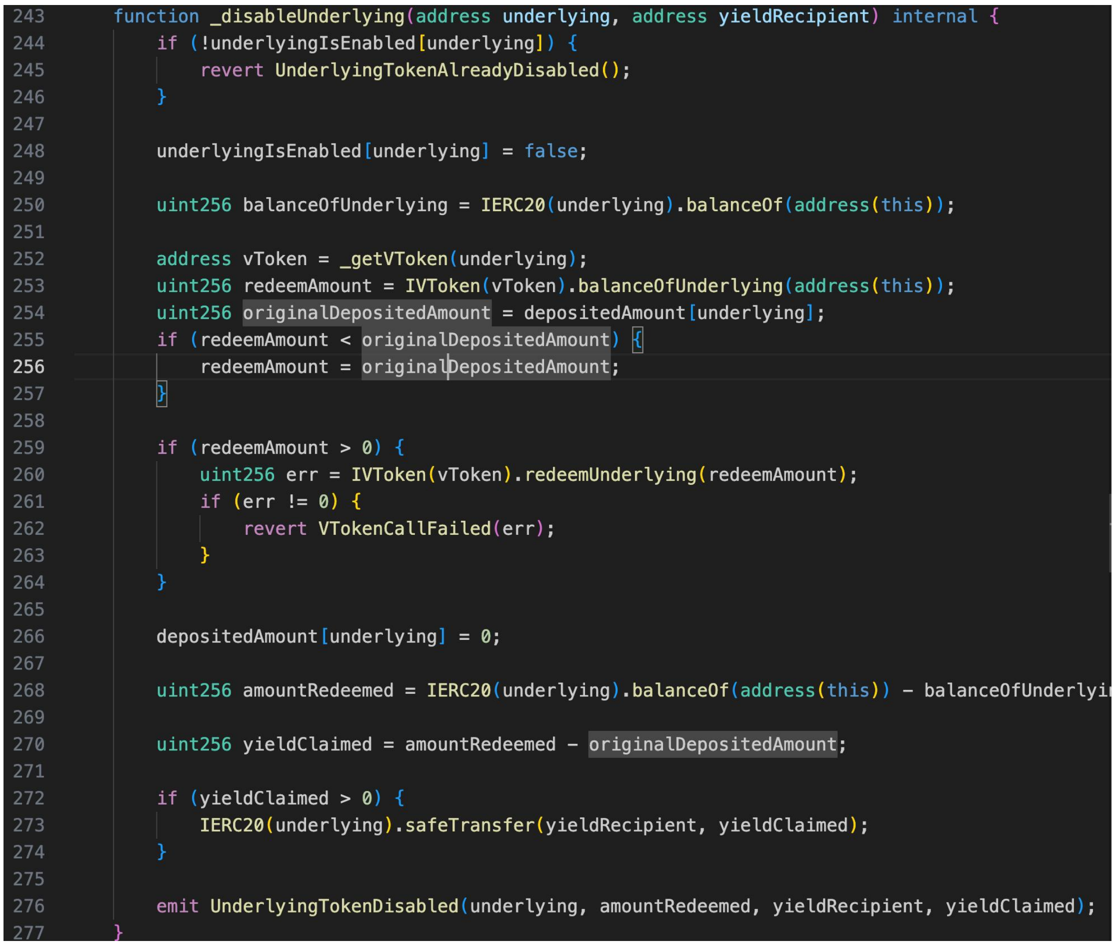
Impact:
Low severity due to privileged access control, but can irreversibly burn yield (recipient=0) or brick yield operations (yieldManager=0).

Recommendation:
Add explicit non-zero checks and revert with clear custom errors.

[L-6] _disableUnderlying may permanently fail when underlying cannot cover recorded principal

Description:
_disableUnderlying sets redeemAmount to at least originalDepositedAmount if balanceOfUnderlying < depositedAmount. If Venus cannot redeem that much (insolvency or liquidity shortage), redeemUnderlying(redeemAmount) returns a non-zero error and the function reverts - preventing disable from succeeding in exactly the emergency scenarios suggested by the comment. The function also computes yieldClaimed = amountRedeemed - originalDepositedAmount, which would underflow if principal is not fully recovered.

YieldBearing/Venus.sol Line#201 Line#256


```
243     function _disableUnderlying(address underlying, address yieldRecipient) internal {
244         if (!underlyingIsEnabled[underlying]) {
245             revert UnderlyingTokenAlreadyDisabled();
246         }
247
248         underlyingIsEnabled[underlying] = false;
249
250         uint256 balanceOfUnderlying = IERC20(underlying).balanceOf(address(this));
251
252         address vToken = _getVToken(underlying);
253         uint256 redeemAmount = IVToken(vToken).balanceOfUnderlying(address(this));
254         uint256 originalDepositedAmount = depositedAmount[underlying];
255         if (redeemAmount < originalDepositedAmount) {
256             redeemAmount = originalDepositedAmount;
257         }
258
259         if (redeemAmount > 0) {
260             uint256 err = IVToken(vToken).redeemUnderlying(redeemAmount);
261             if (err != 0) {
262                 revert VTokenCallFailed(err);
263             }
264         }
265
266         depositedAmount[underlying] = 0;
267
268         uint256 amountRedeemed = IERC20(underlying).balanceOf(address(this)) - balanceOfUnderlyi
269
270         uint256 yieldClaimed = amountRedeemed - originalDepositedAmount;
271
272         if (yieldClaimed > 0) {
273             IERC20(underlying).safeTransfer(yieldRecipient, yieldClaimed);
274         }
275
276         emit UnderlyingTokenDisabled(underlying, amountRedeemed, yieldRecipient, yieldClaimed);
277     }
```

Impact:
Emergency“disable”may be unavailable when needed most.

Recommendation:
Clarify intended semantics (fail-closed to preserve solvency/accounting vs fail-open to recover whatever is possible). If fail-open is desired, add an alternative path that redeems what can be redeemed, records shortfall explicitly, and avoids underflow.

4.2 Informational

[I-1] outcomeSlotCount==256 allowed in prepareCondition but breaks fullIndex Set computation

ConditionalTokens.sol permits outcomeSlotCount <= 256 in prepareCondition, but splitPosition/mergePositions/redeemPositions compute uint fullIndexSet = (1 << outcomeSlotCount) - 1, which reverts for 256 in Solidity 0.8+, so align semantics by capping at 255 or special-casing 256 (e.g., type(uint256).max).

[I-2] Use SafeERC20 consistently for transfer/transferFrom

In ConditionalTokens.sol and YieldBearingConditionalTokens.sol, replace raw ERC20 transfer/transferFrom patterns with OpenZeppelin SafeERC20.safeTransfer/safeTrans ferFrom to improve compatibility with non-standard ERC20s and reduce DoS risk.

[I-3] Use forceApprove (or safe approve pattern) for ERC20 approvals

In Exchange/mixins/Assets.sol (constructor approval to CTF) and UMA/UmaCtfAdapt er.sol::_requestPrice (approval to oracle), prefer SafeERC20.forceApprove (or "set 0 then set max") to avoid failures on tokens requiring allowance resets.

[I-4] Add explicit array-length checks to avoid out-of-bounds reverts

Add require(orders.length == fillAmounts.length) / require(makerOrders.length == ma kerFillAmounts.length)-style checks in Exchange/mixins/Trading.sol::_fillOrders/_fillMa kerOrders and similarly in fee refund loops (ExchangeFee/FeeModule.sol::_refundFee s, ExchangeFee/FeeModuleV2.sol::_refundMakerFees) to fail fast with clear errors.

[I-5] Several comments do not match implemented access control or paramet er meaning

Fix misleading comments: FeeModuleV2._refundTakerFees's receiveAmount description, NegRiskOperator.prepareQuestion comment ("OnlyAdmin" vs onlyOperator), YieldBearingWrappedCollateral.connectVTokenToUnderlying/claimYield ("only owner" vs onlyYieldManager), and ConditionalTokensFeesHandler ("only admins" vs onlyRole(WITHDRAW_AND_MERGE_ROLE)), to reduce operational mistakes.

[I-6] vToken verification uses `isVToken()` only; compromised `YIELD_MANAGE R_ROLE` can cause fund loss

Venus._connectVTokenToUnderlying only checks IVToken(vToken).isVToken() before wiring underlying -> vToken, and yield-bearing flows later forceApprove the configu red vToken and call mint/redeem; therefore, if YIELD_MANAGER_ROLE is compro mised or malicious and points to a fake "vToken", it could exploit approvals/calls t o drain underlying funds. Treat the yield manager as a high-trust role (multisig/time lock/monitoring) and consider on-chain allowlisting/stronger invariant checks (expect ed underlying, known comptroller/market registry) to reduce blast radius.