QuillAudits

AUDIT REPORT

January 2026

For

KrypC

Table of Content

Executive Summary

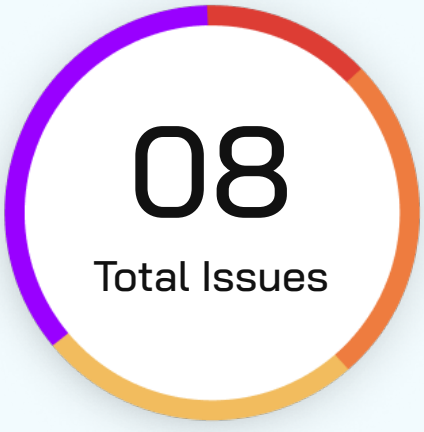
Project Name	KrypC
Protocol Type	Vault
Project URL	https://krypc.com/
Overview	The MbillzUpgradeable contract is an upgradable ERC20-based investment vault that tokenizes user deposits and represents them as shares, enabling proportional ownership over the vault's pooled assets. Built using OpenZeppelin's upgradeable framework, it incorporates UUPS upgradability, pausability, reentrancy protection, and owner-controlled administration. The system manages multiple investment "positions," allowing deposits, withdrawals, profit distribution, and automated tracking of active positions through internal bookkeeping. Share minting and burning follow a valuation-based model to maintain fairness when users enter or exit the pool. Administrative functions allow updating assets, managing strategies, and handling emergency controls, ensuring operational flexibility.
Audit Scope	The scope of this Audit was to analyze the Krypc Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/krypc-code/deltaneutral-contracts
Branch	multi-exchange-support
Contracts in Scope	HyperLiquidExchange.sol MbillzUpgradeableGeneric.sol IDEX,IExchange,IPool interfaces
Commit Hash	9a11dfed11f130d7f7cb275150b483eea08e491f
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	10th December 2025 - 22nd December 2025

Updated Code Received	5th January 2026
Review 2	5th January 2026
Fixed In	04456a23887972d5481d915ba067132acc560c9b

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0 (0.0%)
High	1 (12.5%)
Medium	2 (25.0%)
Low	2 (25.0%)
Informational	3 (37.5%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	**1**	**2**	**2**	**3**

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Division by Zero in Share Calculation Due to getNAV() Returning Zero (Deposit Bricking)	High	Resolved
2	Front-Running Opportunity Around Unrealized PnL Updates Leading to Share Inflation	Medium	Resolved
3	Withdrawals Are Priced at Execution NAV Instead of Queue-Time NAV	Medium	Resolved
4	Signed-to-Unsigned Cast on Negative Pool Value Causes Incorrect User Accounting	Low	Resolved
5	Unbounded Platform Fee Allows Fees to Exceed 100%	Low	Resolved
6	Inconsistent Unit Handling for minDepositAmount Between Initialization and Updates	Informational	Resolved
7	Missing Balance Verification in dexReturnFunds Can Corrupt Pool Accounting	Informational	Resolved
8	Missing Balance Verification in Exchange Adapters Can Corrupt Pool Accounting	Informational	Resolved

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls

- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

Division by Zero in Share Calculation Due to getNAV() Returning Zero (Deposit Bricking)

Resolved

Path

MbillzUpgradeableGeneric.sol

Function Name

`deposit()`

Description

The deposit function calculates minted shares by dividing the deposit amount by the Net Asset Value (NAV). However, getNAV() can legitimately return 0 under multiple conditions. When this happens, the share calculation performs a division by zero, causing a revert and permanently bricking deposits while totalSupply() > 0.

Specifically

**If getLivePoolValue() returns <= 0, getNAV() returns 0.
If totalSupply() becomes larger than poolValue * 1e6, integer division truncation can also result in NAV == 0.**

Once totalSupply() > 0, all deposits rely on getNAV(), meaning a zero NAV fully halts deposits.

This creates a protocol-level DoS, where users cannot deposit funds even though the contract is not paused and remains otherwise functional. The issue is high severity because it can be triggered by market conditions or accounting drift, not just misconfiguration.

```
function deposit(uint256 amount) external nonReentrant whenNotPaused {
    if (!isKYCVerified[msg.sender]) revert("User not KYC-verified");
    if (amount == 0) revert InvalidAmount();
    if (amount < minDepositAmount) revert BelowMinimumDeposit();

    if (!USDC.transferFrom(msg.sender, address(this), amount))
        revert InsufficientBalance();

    uint256 shares;
    if (totalSupply() == 0) {
        shares = amount;
    } else {
        uint256 nav = getNAV();
        // @audit nav can be 0, causing division by zero
        shares = (amount * 1e6) / nav;
    }
}

function getNAV() public view returns (uint256) {
    uint256 supply = totalSupply();
    if (supply == 0) return 1e6;

    int256 poolValue = getLivePoolValue();
    if (poolValue <= 0) return 0;

    return (uint256(poolValue) * 1e6) / supply;
}
```

Impact

A zero NAV permanently disables all future deposits while the protocol remains alive.
This is a protocol-level denial of service affecting capital inflow, share accounting, and long-term solvency.

Likelihood: Medium

POC

Preconditions:

- totalSupply() > 0 (at least one deposit already exists)
- Contract is not paused
- At least one exchange is registered
- NAV is externally updated via navUpdator

Step 1 — Initial healthy state

totalAvailableBalance = 1,000 USDC
totalAllocated = 0
unrealizedPnl = 0
totalSupply = 1,000 shares

NAV calculation:
poolValue = 1,000
NAV = (1,000 * 1e6) / 1,000 = 1e6

Step 2 — Exchange allocation

Fund manager allocates funds:
allocateToExchange(amount = 1,000)

totalAvailableBalance = 0
totalAllocated = 1,000
unrealizedPnl = 0

Step 3 — Unrealized loss update (legitimate MTM)

updateExchangeUnrealizedPnlBatch([exchange], [-1,100])

getLivePoolValue()
= totalAvailable + totalAllocated + unrealizedPnl
= 0 + 1,000 - 1,100
= -100

Step 4 — NAV becomes zero

```
function getNAV() public view returns (uint256) {
    if (poolValue <= 0) return 0;
}
```

Step 5 — User attempts to deposit

A KYC-verified user calls:
- deposit(100 USDC)
- uint256 nav = getNAV(); // nav == 0
- shares = (amount * 1e6) / nav; // division by zero → revert

Step 6 — Permanent DoS condition

At this point:
• totalSupply() >
• NAV == 0
• Deposits always revert
• Contract is not paused
• No on-chain role can fix this immediately

Unless NAV later becomes strictly positive and large enough to avoid truncation, deposits remain permanently bricked.

Recommendation

Prevent deposits when NAV is zero or too small, and enforce a non-zero NAV invariant

Medium Severity Issues

Front-Running Opportunity Around Unrealized PnL Updates Leading to Share Inflation

Resolved

Path

MbillzUpgradeableGeneric.sol

Function Name

`deposit(uint256 amount)`

Description

The protocol allows unrealized PnL values to be updated via the permissioned updateExchangeUnrealizedPnlBatch function, which directly affects the Net Asset Value (NAV) used for share minting during deposits. Because deposits and NAV updates are not synchronized, users can deposit immediately before a positive unrealized PnL update is applied, when the NAV is temporarily lower than its fair value.

Since the number of shares minted is inversely proportional to the NAV, depositing prior to a positive NAV update results in more shares being minted for the same amount of USDC. After the unrealized PnL update is applied, the NAV increases, causing the depositor to hold a disproportionately large share of the pool relative to their contribution. This leads to dilution of existing liquidity providers and creates an economic race condition around NAV updates.

The issue arises because NAV updates are applied discretely, deposits are always permitted, and no freshness checks, epoching, or deposit locks are enforced around valuation updates.

```
function updateExchangeUnrealizedPnlBatch(
    bytes32[] calldata exchangeKeys,
    int256[] calldata unrealizedPnls
) external onlyNavUpdator {
    if (exchangeKeys.length != unrealizedPnls.length)
        revert InvalidAmount();

    for (uint256 i = 0; i < exchangeKeys.length; i++) {
        ExchangeData storage ex = exchanges[exchangeKeys[i]];
        if (!ex.isValid) revert InvalidAddress();

        ex.unrealizedPnl = unrealizedPnls[i];
        emit ExchangeUnrealizedPnlUpdated(
            exchangeKeys[i],
            unrealizedPnls[i]
        );
    }
}
```

Impact

This issue does not lead to direct loss of funds or protocol insolvency; however, it results in unfair share allocation and economic dilution of existing depositors. Over time, repeated exploitation of this timing window can skew ownership distribution and reduce trust in the protocol's share pricing mechanism. The protocol remains functional, but value is redistributed in favor of strategically timed depositors.

Likelihood: Medium

Exploitation requires knowledge of an upcoming positive unrealized PnL update and sufficient timing to deposit before the update is executed. While the NAV updater is permissioned, updates may still be observable in the mempool or predictable threw off-chain coordination. The attack does not require contract-level manipulation but does rely on operational timing and capital availability.

POC

1. The pool has an existing supply of shares and a NAV based on current unrealized PnL.
2. A NAV updater prepares a transaction that will significantly increase unrealized PnL.
3. Before the NAV update transaction is executed, a user deposits funds while NAV is still low.
4. Shares are minted using the lower NAV:
   shares = (amount * 1e6) / nav_before
5. The NAV update transaction executes, increasing unrealized PnL and raising the NAV.
6. The depositor now holds a larger share of the pool than intended, diluting existing LPs.

Recommendation

Consider introducing synchronization mechanisms between NAV updates and deposit execution. Possible mitigations include:

- Enforcing NAV freshness checks for deposits and withdrawals.
- Temporarily pausing deposits during unrealized PnL updates.
- Implementing epoch-based deposits where share minting uses a finalized NAV.
- Applying time-weighted or delayed share minting to smooth valuation changes.

Withdrawals Are Priced at Execution NAV Instead of Queue-Time NAV

Resolved

Path

MbillzUpgradeableGeneric.sol

Function Name

`executePayout(bytes32 requestId)`

Description

Withdrawals in the protocol follow a two-step process where users first queue a payout request and the fund manager later executes it. However, the withdrawal amount is calculated using the NAV at execution time, not the NAV at the time the withdrawal was queued.

Specifically, although the user commits to withdrawing a fixed number of shares when calling queuePayout, the USDC amount they ultimately receive is determined during executePayout using the current NAV. This introduces slippage risk, as the NAV may change between the queue and execution steps.

Because NAV depends on unrealized PnL updates and exchange activity, users are exposed to price movement they cannot control after committing to withdraw. In the worst case, a user may receive significantly less value than expected if NAV decreases between queueing and execution. Conversely, NAV increases could also benefit the user, making outcomes unpredictable.

Impact

This issue does not lead to direct loss of funds or protocol insolvency; however, it results in unfair share allocation and economic dilution of existing depositors. Over time, repeated exploitation of this timing window can skew ownership distribution and reduce trust in the protocol's share pricing mechanism. The protocol remains functional, but value is redistributed in favor of strategically timed depositors.

Likelihood: Medium

Users cannot predict the final withdrawal amount at the time of queueing.
Users are exposed to slippage and potential value loss if NAV declines before execution.

POC

1. A user queues a withdrawal for a fixed number of shares when NAV is high.
2. Before executePayout is called, unrealized PnL is updated negatively, reducing NAV.
3. The fund manager executes the payout using the lower NAV.
4. The user receives fewer USDC than expected at the time of queueing, despite having already committed their shares.

Recommendation

Consider pricing withdrawals using the NAV at queue time rather than execution time.

Low Severity Issues

Signed-to-Unsigned Cast on Negative Pool Value Causes Incorrect User Accounting

Resolved

Path

MbillzUpgradeableGeneric.sol

Function Name

`getUserPosition()`

Description

The getUserPosition function derives a user's currentValue from getLivePoolValue(), which returns an int256. When the pool value is negative, the function performs arithmetic on signed integers and then casts the result to uint256 without validating that the value is non-negative. In Solidity, casting a negative int256 to uint256 results in a very large number due to two's-complement wrapping.

As a result, when the pool value is negative, currentValue can be reported as an extremely large positive number. This in turn affects the calculated netProfitLoss, which may flip from a loss to a profit or overflow and revert due to the inflated intermediate value.

This issue occurs in a view-only function and does not affect protocol state, but it can lead to incorrect user-facing accounting and misleading information for off-chain consumers.

Impact

Low as it only affect view only function

Likelihood: Low to Medium

Recommendation

Explicitly handle negative pool values before performing unsigned casts.

Unbounded Platform Fee Allows Fees to Exceed 100%

Resolved

Path

MbillzUpgradeableGeneric.sol

Function Name

`updatePlatformFee(uint256 _feeBps)`

Description

The updatePlatformFee function allows the contract owner to set platformFeeBps without enforcing an upper bound. Because fees are applied during withdrawal execution, setting the platform fee above 10,000 basis points (100%) can result in withdrawals being partially or fully consumed by fees

Impact

The function is restricted to the contract owner, and exploitation requires either misconfiguration or malicious behavior by a privileged role. However, configuration errors are realistic, especially during upgrades or operational changes.

Likelihood: Medium

Recommendation

Enforce an upper bound on the platform fee to ensure it cannot exceed 100%:

Informational Issues

Inconsistent Unit Handling for minDepositAmount Between Initialization and Updates

Resolved

Path

MbillzUpgradeableGeneric

Function Name

`updatePoolParameters()`

Description

During contract initialization, the minDepositAmount value is scaled by 10^6 to account for USDC decimals
However, when updating the pool parameters via updatePoolParameters, the new minimum deposit amount is assigned directly without applying the same scaling
This results in inconsistent units for minDepositAmount depending on whether the value was set during initialization or updated later. As a consequence, the minimum deposit requirement may unintentionally become significantly higher or lower than intended after an update.

Recommendation

Ensure consistent unit handling for minDepositAmount across initialization and updates.

Missing Balance Verification in dexReturnFunds Can Corrupt Pool Accounting

Resolved

Path

MbillzUpgradeableGeneric

Function Name

`dexReturnFunds(uint256 amount)`

Description

The dexReturnFunds function increases totalAvailableBalance based solely on the amount parameter provided by the DEX contract, without verifying that the corresponding amount of USDC has actually been transferred back to the pool.

The function assumes that the DEX has already transferred amount USDC to the pool contract, but this assumption is not validated by checking the actual USDC balance before and after the call. As a result, the internal accounting (totalAvailableBalance) can become inconsistent with the real USDC balance held by the contract.

The function is restricted to the registered DEX contract, reducing exposure to untrusted callers. However, accounting mismatches can realistically occur due to integration bugs, upgrade mistakes, or unexpected execution paths in the DEX contract. No malicious behavior is required.

Recommendation

Verify that the pool's USDC balance has increased by the expected amount before updating internal accounting.

Missing Balance Verification in Exchange Adapters Can Corrupt Pool Accounting

Resolved

Path

HyperliquidExchange

Function Name

`transferFundsBackToPool()`

Description

HyperliquidUpgradeable adapter contract update pool accounting based on assumed token transfers without verifying that the expected amount of USDC was actually received by the pool.

In both adapters, the fund return flow follows this pattern:

```
USDC.transfer(pool, returnedAmount);
IPool(pool).closeExchangeAllocation(...);
```

The adapters assume that returnedAmount USDC has been successfully transferred to the pool, but they do not verify the pool's USDC balance before and after the transfer. As a result, the pool finalizes allocations and updates internal accounting based on potentially incorrect assumptions.

Similarly, during allocation, both adapters forward funds without validating balance changes, relying entirely on successful ERC20 calls and trusted off-chain behavior.
This creates an accounting integrity risk where the pool's internal state may diverge from the actual USDC balance held, especially in the presence of integration bugs, non-standard ERC20 behavior, partial transfers, or operational errors.

The affected functions are permissioned and rely on trusted operators (Fund Manager and exchange adapters). However, accounting mismatches can realistically occur due to integration mistakes, future upgrades, or unexpected ERC20 behavior.

Recommendation

Introduce explicit balance verification when funds are returned to the pool, or redesign the flow so that the adapter itself performs the token transfer and accounting update atomically.

At minimum:
- Use SafeERC20 for all token transfers.
- Validate that the pool's USDC balance increases by at least returnedAmount before finalizing the allocation.

Functional Tests

- Should initialize contracts correctly
- Should allow allocation only from pool
- Should transfer funds back to pool on fund return
- Should revert fund return if pool address is invalid
- Should allow only fund manager to update KYC status
- Should allow deposit only above minimum deposit amount
- Should allow user to queue payout request
- Should burn user shares after payout
- Should update realized profit and loss correctly
- Should allow owner to pause the contract

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats

	deposit(uint256 amount)	Allows a KYC-verified user to deposit USDC into the pool and receive pool shares based on current NAV.
		Inputs
		amount
		• Control: Fully controlled by the user.
		• Constraints:
		  • Must be non-zero
		  • Must be ⩾ minDepositAmount
		  • User must be KYC-verified
		  • User must have sufficient USDC balance and allowance
		• Impact:
		  • Determines the amount of capital added to the pool and the number of shares minted.
		Threats & Risks
		• **NAV manipulation risk:** Shares are minted using the current NAV, which depends on externally supplied unrealized PnL.
		• **Division-by-zero risk:** If getNAV() returns zero while totalSupply > 0, deposits revert (protocol-level DoS).
		• **Rounding / inflation risk:** Incorrect NAV scaling or truncation can cause share mispricing.
		• **Front-running risk:** Deposits can be strategically timed around NAV updates to gain favorable pricing.

Contract	Function	Threats
		Intended Branches • Should revert if user is not KYC verified. • Should revert if deposit amount is below minimum. • Should mint shares equal to deposit amount on first deposit. • Should mint shares proportionally based on NAV for subsequent deposits. **Negative Behavior** • Should not allow deposit when NAV is zero. • Should not allow deposit if contract is paused. • Should not allow deposit without sufficient USDC balance or allowance.
	queuePayout(address investor, uint256 shares, bytes32 requestId)	Queues a withdrawal request for a user without immediately transferring funds. **Inputs** **investor** • Control: User-controlled. • Constraints: Must equal msg.sender. • Impact: Identifies the withdrawing user. **shares** • Control: User-controlled. • Constraints: Must be > 0 and ⩽ user share balance. • Impact: Determines withdrawal size.

Contract	Function	Threats
		shares • Control: User-controlled. • Constraints: Must be unique. • Impact: Prevents replay or overwriting of requests. **Threats & Risks** • **State locking risk:** Shares are not locked at queue time; user retains full control until execution. • **Execution timing risk:** Withdrawal value is not fixed at queue time. **Intended Branches** • Should create a new payout request. • Should revert if requestId already exists. • Should revert if shares exceed balance.
	executePayout(bytes32 requestId)	Executes a queued withdrawal request and transfers USDC to the user. **Inputs** **requestId** • Control: Fund Manager controlled. • Constraints: Must reference a valid, unexecuted request. • Impact: Determines which withdrawal is executed.

Contract	Function	Threats
		Threats & Risks • Slippage risk: Withdrawal amount is • priced at execution-time NAV, not queue-time NAV. • NAV manipulation risk: NAV updater can influence withdrawal value prior to execution. Liquidity risk:Withdrawal can revert if totalAvailableBalance is insufficient. **Intended Branches** • Should revert if request is invalid or already executed. • Should burn user shares on execution. • Should deduct platform fees correctly. • Should transfer net USDC to the user. **Negative Behavior** • Should not allow non-fund-manager execution. • Should not allow execution when pool lacks liquidity.
	allocateToExchange(bytes32 exchangeKey, uint256 amount, bytes data)	Allocates pool funds to an external exchange via adapter contracts. **Inputs** **exchangeKey** • Control: Fund Manager controlled. • Constraints: Must reference a registered exchange. • Impact: Selects destination for capital deployment.

Contract	Function	Threats
		amount • Control: Fund Manager controlled. • Constraints: Must be ⩽ available • balance and within utilization limits. Impact: Reduces pool liquidity and increases deployed capital. **Threats & Risks** • Liquidity exhaustion risk: Incorrect utilization logic can allow excessive allocation. • Accounting trust risk: Pool assumes adapter transfers funds correctly. • Centralization risk: Fund Manager fully controls deployment timing and size. **Intended Branches** • Should revert if utilization exceeds maxPoolUtilization. • Should reduce available balance. • Should increase exchange allocation.
	closeExchangeAllocation(...)	Finalizes an exchange allocation and records realized PnL. **Inputs** **returnedAmount** • Control: Exchange adapter controlled. • Constraints: None enforced on-chain. • Impact: Increases pool available balance.

Contract	Function	Threats
		realizedPnl • Control: Exchange adapter • controlled. • Constraints: Informational only. Impact: Updates performance metrics. **Threats & Risks** • Accounting mismatch risk: Returned amount is trusted without balance verification. • PnL misreporting risk: Realized PnL does not affect NAV directly and can diverge from reality. **Intended Branches** • Should update available balance. • Should update realized profit/loss metrics. • Should reduce exchange allocation.
	updateExchangeUnrealizedPnlBatch(...)	Updates unrealized PnL values used for NAV calculation. **Inputs** **unrealizedPnls** • Control: NAV Updater controlled. • Constraints: None. • Impact: Directly affects NAV and share pricing.

Contract	Function	Threats
		Threats & Risks • NAV manipulation risk: Single update • can significantly change NAV. Timing attack risk: Deposits and • withdrawals can be timed around updates. Trust risk: No on-chain validation of correctness. **Intended Branches** • Should update unrealized PnL for valid exchanges. • Should revert on mismatched array lengths.

Closing Summary

In this report, we have considered the security of KrypC Smart Contracts. We performed our audit according to the procedure described above.

1 High, 2 Medium, 2 Low and 3 Informational severity bugs were found, which have been noted and resolved by team.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

January 2026

For

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com