QuillAudits

Audit Report
November, 2024

For

pluto

Table of Content

Table of Content

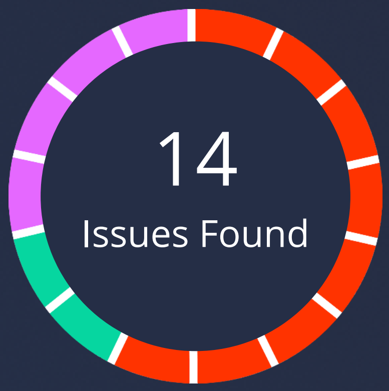
Executive Summary

Project Name	Pluto
Overview	Pluto is a decentralized finance (DeFi) platform on the Solana blockchain, offering advanced yield optimization tools. It provides a user-friendly, one-click yield optimizer that delivers top yields, innovative strategies, and leveraged yield farming, all while allowing users to manage their risk exposure effectively.
Timeline	04th November 2024 - 24th November 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The scope of this audit was to analyze the contracts in scope for quality, security, and correctness.
Blockchain	Solana
Source Code	https://github.com/plutoleverage/pluto-so-c f812be80ca8dcac52f96229414d705a91018992c

Executive Summary

Contracts In-Scope	handlers/handler_protocol_create.rs handlers/handler_protocol_set.rs handlers/handler_earn_config_create.rs handlers/handler_earn_config_set.rs handlers/handler_vault_earn_create.rs handlers/handler_vault_earn_set_index.rs handlers/handler_vault_earn_deposit.rs handlers/handler_vault_earn_withdraw.rs handlers/handler_leverage_config_create.rs handlers/handler_leverage_config_set.rs handlers/handler_vault_leverage_create.rs handlers/handler_vault_leverage_set_index.rs handlers/handler_vault_leverage_fund.rs handlers/handler_vault_leverage_confiscate.rs handlers/handler_vault_leverage_close.rs handlers/handler_vault_leverage_release.rs handlers/handler_vault_leverage_repay_borrow.rs handlers/handler_vault_leverage_closing.rs handlers/handler_vault_leverage_safe.rs handlers/handler_vault_leverage_eject.rs handlers/handler_vault_leverage_liquidate.rs handlers/handler_vault_leverage_take_profit.rs handlers/handler_vault_leverage_keeper_release.rs handlers/handler_vault_leverage_keeper_repay_borrow.rs handlers/handler_vault_leverage_keeper_closing.rs handlers/handler_vault_leverage_keeper_pay_liquidation_fee.rs handlers/handler_wrap_sol.rs handlers/handler_unwrap_sol.rs utils/decimals.rs utils/fraction.rs
Branch	upgrade/event/v2
Fixed In	https://github.com/plutoleverage/pluto-so-c/tree/fixing-audit/v1

Number of Security Issues per Severity



🟥 High	🟨 Medium
🟩 Low	🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	3	0	1	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	5	0	1	4

Checked Vulnerabilities

We have scanned the solana program for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- ✓ Signer authorization
- ✓ Account data matching
- ✓ Sysvar address checking
- ✓ Owner checks
- ✓ Type cosplay
- ✓ Initialization
- ✓ Arbitrary cpi
- ✓ Duplicate mutable accounts
- ✓ Bump seed canonicalization
- ✓ PDA Sharing
- ✓ Incorrect closing accounts
- ✓ Missing rent exemption checks
- ✓ Arithmetic overflows/underflows
- ✓ Numerical precision errors
- ✓ Solana account confusions
- ✓ Casting truncation
- ✓ Insufficient SPL token account verification
- ✓ Signed invocation of unverified programs

Techniques and Methods

Throughout the audit of Pluto, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of various token standards.
- 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 Solana programs.

Structural Analysis

In this step, we have analysed the design patterns and structure of Solana programs. A thorough check was done to ensure the Solana program is structured in a way that will not result in future problems.

Static Analysis

Static analysis of Solana programs was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of Solana programs.

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 analysed, and 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 behaviour of Solana programs in production. Checks were done to know how much gas gets consumed and the possibilities of optimising code to reduce gas consumption.

Types of Severity

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

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

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.

High Severity Issues

1. Missing Confidence Interval Handling in Pyth Price Feeds

Description

The protocol uses Pyth's get_price_no_older_than function in several core functionalities, including VaultLeverageClose, VaultLeverageEject, VaultLeverageSafe, VaultLeverageFund, VaultLeverageLiquidate, VaultLeverageSetIndex, and VaultLeverageTakeProfit. However, the confidence interval (μ±σ) provided by Pyth, which defines the range where the true price likely lies with 95% probability, is ignored in these implementations. The lack of confidence interval consideration leads to a narrow interpretation of price accuracy and leaves the protocol exposed to volatile or uncertain market conditions. Confidence intervals provide critical safeguards against price uncertainty during market turbulence or price manipulation, allowing protocols to adjust valuations conservatively to prevent over-leveraging or unnecessary liquidations.

Impact

1. VaultLeverageFund: Collateral deposited by the user is valued using the aggregate price (μ), ignoring the lower bound (μ−σ). The collateral may be overvalued, enabling users to borrow or leverage beyond the safe limit. This increases the risk of bad debt and protocol insolvency if the market moves unfavorably.
2. VaultLeverageClose When users close positions, the debt repayment and collateral release calculations are based solely on μ, without considering (μ+σ). Under extreme volatility, the protocol may underestimate debt, leading to insufficient repayment. This causes loss to the protocol and exposes it to insolvency risks.
3. VaultLeverageLiquidate Liquidations are triggered when the collateral value falls below the debt, using μ for both calculations. Ignoring μ−σ for collateral valuation may delay liquidations, while ignoring μ+σ for debt valuation may prematurely liquidate positions.
4. VaultLeverageTakeProfit Profits are calculated based on μ, without considering (μ±σ). Inaccurate profit calculations can lead to incorrect valuations of user gains, reducing user trust.

Remediation

Incorporate Confidence Intervals:
- Collateral Valuation: Use $\mu - \sigma$ for conservative collateral valuations.
- Debt Valuation: Use $\mu + \sigma$ during debt calculations for liquidations and repayments.
- Profit Valuation: Use $\mu + \sigma$ to minimize premature take-profit scenarios.

Dynamic Risk Adjustment:
- Introduce thresholds based on the confidence ratio (σ/μ). For example, if $(\sigma/\mu) > 5\%$, pause user actions (e.g., position opening) until the confidence improves.

Status

Acknowledged

2. Borrowers Face Immediate Liquidation Risk After Protocol Freeze is Lifted

Description

The protocol's check_freeze mechanism halts both repayment and liquidation operations during a freeze, ensuring that no transactions are processed. This freeze is designed to safeguard the protocol in exceptional scenarios, but it unintentionally creates an issue for borrowers. When the protocol is frozen: - Borrowers are unable to repay their debts and improve their health factors. - Liquidators cannot liquidate over-leveraged positions. During the freeze, external market fluctuations (e.g., price drops in collateral assets) can worsen borrowers' health factors, potentially pushing them below the liquidation threshold. Once the freeze is lifted, these vulnerable positions become immediate targets for liquidation. Borrowers are left at a significant disadvantage, unable to preemptively secure their positions due to the enforced freeze.

Impact

1. Borrowers lose collateral due to liquidation triggered by market changes during the freeze.
2. Borrowers are unable to repay during the freeze, leading to unfair losses once operations resume.
3. Liquidators gain an undue advantage, able to immediately target vulnerable positions post-freeze.
4. Participants may perceive the protocol as unfair, discouraging future engagement by borrowers and lenders.

Remediation

After the freeze is lifted, enforce a grace period where borrowers cannot be liquidated but can repay or adjust their collateral. This duration could match the length of the freeze, capped at a reasonable maximum (e.g., 24 hours).

Status

Acknowledged

3. Liquidation Fee Bypass via Improper Instruction Sequencing

Target

src/handlers/handler_vault_leverage_keeper_repay_borrow.rs#L84-93

Description

The protocol's liquidation mechanism involves several sequential instructions, such as LeverageVaultKeeperRepayBorrow, LeverageVaultKeeperPayLiquidationFee, and LeverageVaultKeeperClosing. These instructions handle key operations such as repaying borrow amounts, paying liquidation fees, and closing positions. If the correct sequence of instructions is not enforced, liquidation fees can be bypassed in two scenarios

Scenario 1: Keeper Calls LeverageVaultKeeperClosing Before LeverageVaultKeeperPayLiquidationFee The LeverageVaultKeeperRepayBorrow function is executed to repay the borrow amount. Instead of calling LeverageVaultKeeperPayLiquidationFee, the keeper directly invokes LeverageVaultKeeperClosing. This prematurely closes the position and skips the liquidation fee payment step. Result: Liquidators are not rewarded, and the protocol loses its liquidation fee revenue.

Scenario 2: Emergency Eject Skipping Liquidation Fees If an emergency eject is initiated, the VaultLeverageEject function allows safe closure of a position. Even if the health factor drops below 1, the emergency eject bypasses the LeverageVaultKeeperPayLiquidationFee. This results in a scenario where the position is safely closed without compensating the liquidator or the protocol. Result: The liquidation fee mechanism is bypassed, leading to economic inefficiencies and potential exploitation.

Impact

- The protocol loses the revenue generated from liquidation fees.
- Liquidators are not incentivized to perform liquidations, potentially destabilizing the protocol.

Remediation

1. Enforce Sequential Execution of Instructions - Add a validation step in LeverageVaultKeeperClosing to ensure that LeverageVaultKeeperPayLiquidationFee has been executed before the position is closed.
2. Restrict Emergency Eject When HF < 1 - Modify VaultLeverageEject to prevent execution if the health factor is below 1. Ensure liquidation fees are processed before emergency eject proceeds

Status

Resolved

4. Misleading Index Growth Due to Time-Based Updates in Earn and Leverage Vaults

Target

src/state/vault_leverage.rs#L191-192, src/state/vault_earn.rs#L316

Description

The update_rate function is invoked by calling the functions earn_vault_set_index & leverage_vault_set_index and has the following code respectively.

```
// earn_vault_set_index()
pub fn set_index(&mut self, last_index_updated: i64, index: u128, apy: u32) -> Result<()> {
    require!(index > 0, Errors::InvalidAmountZero);
    self.index = index;
    self.last_index_updated = last_index_updated;
    self.apy.update_rate(apy, self.last_index_updated)?;
    Ok(())
}
```

Here, the update_rate function updates the APY rate and calculates the exponential moving averages (EMAs) over time intervals like hourly, daily, and beyond, based on the last_index_updated timestamp.

```
// leverage_vault_set_index()
pub fn set_index(&mut self, last_index_updated: i64, index: u128, apy: u32,
borrowing_index: u128, borrowing_apy: u32) -> Result<()> {
    require!(index > 0, Errors::InvalidAmountZero);
    require!(borrowing_index > 0, Errors::InvalidAmountZero);
    self.last_index_updated = last_index_updated;
    self.index = index;
    self.borrowing_index = borrowing_index;
    self.apy.update_rate(apy, self.last_index_updated)?;
    self.borrowing_apy.update_rate(borrowing_apy, self.last_index_updated)?;
    self.update_time()?;
    Ok(())
}
```

This function similarly calls update_rate for the APR (apy) and borrowing APR (borrowing_apy), ensuring that both indices grow in alignment with time and perceived profitability.
In both contexts, the indices (earn and leverage) grow based on time rather than actual yield generated by supply or borrow activities. This could create an inflated perception of performance, misleading users into believing the vault is generating yield when it is not.

Remediation

Adjustments to leverage_vault_set_index & earn_vault_set_index and should be made to reflect the true performance of the vault in terms of borrowing activity and real returns.

Status

Acknowledged

Pluto Team Comment

Due to the nature of how our dapp work, user (earn lender) borrowed asset are swapped into JLP, INF upon each leverage position opened. Unlike regular borrow lending, where index solely dictated by the supply/borrow ratio, in pluto we calculate the index based on APY of the JLP/INF , which reward gain is included in JLP/INF Price

5. Incorrect Implementation of div_floor, div_ceil, mul_floor, and mul_ceil

Target

programs/pluto/src/util/decimals.rs

Description

The functions div_floor, div_ceil, mul_floor, and mul_ceil are designed to perform fractional arithmetic while applying either flooring or ceiling logic. However, upon closer inspection, the current implementation relies entirely on Rust's default behavior for division and multiplication without explicitly enforcing flooring or ceiling logic.

No Explicit Logic for Floor or Ceil: The functions div_floor and div_ceil use the same logic and rely on Fraction::checked_div, which inherently performs truncation (flooring) in Rust for integers. There is no additional logic to enforce ceiling behavior. Similarly, mul_floor and mul_ceil behave identically because there is no logic to enforce rounding up or down.

Potential for Incorrect Results: In cases where explicit ceiling behavior is required (e.g., rounding up fractional results), the current implementation will fail to deliver the expected results as the entire protocol is heaviliy dependednt on it.

Recommendation

The fixed library provides built-in methods for explicit ceiling and flooring operations on fixed-point numbers, which aligns well with the intended functionality of these functions. It is recommended to adjust the functions to use along with the intended scaling functionality.

1. For Ceil Function: https://docs.rs/fixed/latest/fixed/struct.FixedU128.html#method.ceil
2. For Floor Function: https://docs.rs/fixed/latest/fixed/struct.FixedU128.html#method.floor

Status

Resolved

6. Incorrect Calculation of Thirty-Minute APR

Target

programs/pluto/src/handlers/handler_vault_earn_set_index.rs , programs/pluto/src/handlers/handler_vault_leverage_set_index.rs#L31

Description

The function handle in earn_vault_set_index & leverage_vault_set_index calculates the thirty_minute_apr based on the hourly_apr. This value is derived from higher APR values (e.g., hourly APR or daily APR) and is used in subsequent computations for updating vault indices. However, in both cases, the formula used to compute the thirty_minute_apr is flawed. The error affects downstream computations such as gain, borrow_gain resulting in incorrect indices.

```
let thirty_minute_apr = decimals::div_ceil(
    INDEX_DECIMALS,
    hourly_apr,
    INDEX_DECIMALS,
    3,
    0)?;
```

Remediation

Update the calculation of thirty_minute_apr to use the correct factor of 2.

```
let thirty_minute_apr = decimals::div_ceil(
    INDEX_DECIMALS,
    hourly_apr,
    INDEX_DECIMALS,
    2, // 30 minutes is half an hour
    0,
)?;
```

Status

Resolved

7. Incorrect time_diff Calculation Prevents EMA Updates in Rate Struct

Target

programs/pluto/src/state/rate.rs#L62-65

Description

The update_rate function in the Rate struct is indirectly called through the set_index methods in both earn_vault_set_index and leverage_vault_set_index. However, due to the bug in update_rate where time_diff always evaluates to 0, the EMA calculations for both APY and borrowing APY will fail to update, rendering the functionality ineffective

This can have the following implications:
1. Stale Rate Metrics: Both the hourly and daily EMA values remain uninitialized or outdated, misrepresenting the APY and borrowing APY trends.
2. Protocol Inaccuracy: Any logic depending on these rates, such as user-facing metrics or borrowing/lending terms, could behave incorrectly or mislead users.

```
pub fn update_rate(&mut self, value: u32, timestamp: i64) -> Result<()> {
    self.last_value = value;
    self.last_updated = timestamp;
    // Here timestamp & self.last_updated will evaluted as equal resulting 0.
    let time_diff = timestamp - self.last_updated;
    if time_diff >= constant::TIME_ONE_HOUR {
        ...
    }
 }
```

Remediation

Fix the update_rate function by calculating time_diff before updating self.last_updated. This ensures the correct time difference is used in conditional checks for hourly and daily updates.

```
pub fn update_rate(&mut self, value: u32, timestamp: i64) -> Result<()> {
    let time_diff = timestamp - self.last_updated; // Calculate time difference first
    self.last_updated = timestamp;              // Then update the last_updated timestamp
    ...
}
```

Status

Resolved

8. Missing Validation on leverage_vault in earn_vault_set_index Function Allows Arbitrary Data Loading

Target

programs/pluto/src/handlers/handler_vault_earn_set_index.rs#L154-155

Description

The function handle in earn_vault_set_index operates on multiple accounts, including the leverage_vault. The leverage_vault account is passed into the function, and data is loaded from it to compute important metrics such as leverage_apy and other vault-related calculations The leverage_vault account lacks proper validation, which means any arbitrary vault can be passed as the leverage_vault parameter. If a malicious or incorrect vault is passed, the function will load and operate on unintended or invalid data, leading to incorrect calculations, protocol inconsistencies.

Remediation

Add an explicit validation to ensure the leverage_vault is linked to the earn_config or the vault account. This can be achieved by:

```
pub fn handle(ctx: Context<VaultEarnSetIndex>) -> Result<()> {
   let leverage_vault = ctx.accounts.leverage_vault.load()?;
   let borrow_vault = ctx.accounts.vault.load()?;
   let earn_config = ctx.accounts.earn_config.load()?;
   // Validate that leverage_vault.borrow_vault is the same as vault
   require_keys_eq!(
      leverage_vault.borrow_vault,
      ctx.accounts.vault.key(),
      Errors::InvalidBorrowVault
   );
   // Validate that vault.earn_config is the same as the earn_config in context
   require_keys_eq!(
      borrow_vault.earn_config,
      ctx.accounts.earn_config.key(),
      Errors::InvalidEarnConfig
   );
   // Existing functionality...
   Ok(())
}
```

Status

Resolved

Low Severity Issues

9. Users Can Bypass Fees by Depositing Slightly Less Than the Minimum Fee Threshold

Target

programs/pluto/src/handlers/handler_vault_earn_deposit.rs#L65-86

Description

The earn_vault_deposit function in the provided Solana program is responsible for handling the deposit of tokens into an earnings vault. This function is called via the handler_vault_earn_deposit function, which implements the core logic. The current implementation allows users to deposit amounts slightly less than the threshold where the fee calculation would yield a non-zero value. This leads to a fee = 0 scenario, allowing the user to bypass fees entirely. By depositing in such small increments repeatedly, users can unfairly accumulate vault units without contributing to the fee pool.

Recommendation

Introduce a protocol-wide minimum deposit independent of the total amount that ensures all deposits are subject to fees.

Status

Acknowledged

10. Event Emission in earn_vault_withdraw can be Optimized

Target

programs/pluto/src/handlers/handler_vault_earn_withdraw.rs#L136-170

Description

The function earn_vault_withdraw emits two events, EventEarnWithdraw and EventEarnWithdrawn, which contain overlapping information. This redundancy increases transaction size and storage costs on Solana.

Recommendation

Combine the information from EventEarnWithdraw and EventEarnWithdrawn into a single event structure and emit it once. This will save resources while retaining all necessary information.

Status

Resolved

Informational Issues

11. Unnecessary Lamport Balance Check in unwrap_sol Function

Target

programs/pluto/src/handlers/handler_unwrap_sol.rs#L18

Description

The function unwrap_sol contains an unnecessary check for the user's SOL lamport balance:

require_gte!(user.lamports(), amount, Errors::InsufficientFunds);
This check is irrelevant as the function only deals with WSOL tokens and their associated accounts, not the user's lamport balance.

Remediation

Remove the redundant lamport balance check to streamline the function and avoid confusion.

Status

Resolved

12. Protocol Fees in earn_config_set not validated

Target

programs/pluto/src/state/earn_config.rs#L188-196

Description

The earn_config_set function does not validate critical fee parameters (protocol_fee, deposit_fee, withdraw_fee, and borrow_fee). Without these validations, it is possible to set unreasonable values (e.g., extremely high or negative fees), which could disrupt the protocol and lead to misconfigurations.

Remediation

Added bounds validation for protocol_fee, deposit_fee, withdraw_fee, and borrow_fee to ensure they are reasonable.

Status

Resolved

13. Parameter Naming for Price Oracle Functions Can Be Improved to Enhance Clarity

Target

programs/pluto/src/lib.rs

Description

The functions leverage_vault_create, leverage_vault_change_price_oracle, and earn_vault_change_price_oracle contain misleading parameter names, causing confusion:

1. leverage_vault_create: The parameters token_collateral_decimal and native_collateral_decimal in handler_vault_leverage_create::handle imply they are decimals but actually represent price feed data for token and native collateral.
2. leverage_vault_change_price_oracle: The parameters token_collateral_decimal and native_collateral_decimal in handler_vault_leverage_change_price_oracle::handle also suggest decimals but are price feed data.
3. earn_vault_change_price_oracle: The parameter token_decimal in handler_vault_earn_change_price_oracle::handle implies a decimal value but is actually price feed data for a token.

Status

Resolved

14. Refactoring update_rate can improve code readability & save gas

Target

programs/pluto/src/state/rate.rs#L

Description

The update_rate function in the provided code has a high level of redundant operations and repetitive calculations that can be optimized to save storage and compute resources when deployed on Solana

Remediation

Refactor the EMA update logic into a reusable function:

```
fn update_ema(&self, current_value: u32, previous_ema: u32, period: u64) -> Result<u32> {
    Fraction::from_num(current_value)
        .checked_mul(Self::alpha(period))
        .ok_or(MathOverflow)?
        .checked_add(
            Fraction::from_num(previous_ema)
                .checked_mul(Self::counter_alpha(period))
                .ok_or(MathOverflow)?,
        )
        .ok_or(MathOverflow)?
        .to_num()
}
```

Use this function in update_rate:

```
self.ema_3d = self.update_ema(value, self.ema_3d, 3)?;
self.ema_7d = self.update_ema(value, self.ema_7d, 7)?;
self.ema_14d = self.update_ema(value, self.ema_14d, 14)?;
```

Status

Resolved

Closing Summary

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

14 issues were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture. In the End, Pluto Team Resolved most of the Issues and Acknowledged other issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Pluto smart contract. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Pluto smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Pluto to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

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 six years of expertise, we've secured over 1000 projects globally, averting over $30 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.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
November, 2024

For

pluto

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com