QuillAudits

AUDIT REPORT

June 2025

For

raga

Table of Content

Executive Summary 04

Number of Security Issues per Severity 06

Summary of Issues 07

Checked Vulnerabilities 09

Techniques and Methods 11

Types of Severity 13

Types of Issues 14

Severity Matrix 15

Critical Severity Issues 16

1. Incorrect Staking of Accumulated Fees in Deposit 16

High Severity Issues 17

1. User Funds Become Permanently Inaccessible After emergencyWithdraw Due to Missing Idle Balance Redemption Logic 17

2. while calling withdrawAll() , all fees accumulated also being transferred to vault, leading to 0 fees paid to treasury 18

3. totalAssets value in RagaVault will inflated due to the Accumulated fees on Strategies 19

4. Missing Access Control in setVault Function 20

5. Incorrect Logic in Panic Function. 21

Medium Severity Issues 22

1. Incorrect JoinKind used in _buildHoneyByUsdLp -> _joinReq – single-token join bug 22

2. Incorrect Swap Formula in InfraredStrategy 24

3. Dimension Mismatch of amounts array in InfraredStableStrategy::_joinReq Function 26

Table of Content

4. Attacker can make the RagaVault::mint function unusable with Dust amounts 28

5. Fee Validation Logic Error. 29

🟧 **Low Severity Issues** 30

1. Critical Griefing Attack via Share Price Inflation due to Missing _decimalsOffset() Override in ERC4626Upgradeable Vault 30

2. usage of slot0, for price fetch and its usage is prone to price manipulation 31

3. Incorrect Function Visibility in InfraredStableStrategy 32

4. Missing Zero Address Validation for Admin 33

🟪 **Informational Issues** 34

1. Unsafe Direct approve Usage in BerapawStrategy Causes Potential Staking Failures and Incompatibility with Some ERC–20 34

2. Missing nonReentrant modifier in withdrawAll function 35

Functional Tests 36

Threat Model 37

Automated Tests 41

Closing Summary & Disclaimer 41

Executive Summary

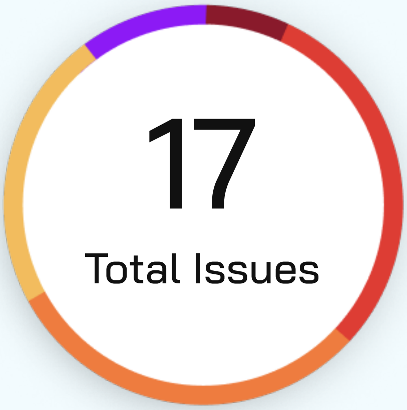
Project Name	Raga Finance
Protocol Type	Vault
Project URL	https://www.raga.finance/
Overview	Raga Finance is a yield aggregation platform that operates through a system of vaults and strategies. The platform allows users to deposit assets into ERC4626-compliant vaults that delegate yield-generating activities to their associated strategy contracts
Audit Scope	The scope of this Audit was to analyze the Raga Finance Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/Nexus-2023/NativeYield/tree/7bd995827d954db0be7c3db6939a0f492e47f7c7
Branch	main
Contracts in Scope	src/vaults/RagaVaultRegistry.sol src/vaults/RagaVault.sol src/strategies/Berachain/BaseStrategy.sol src/strategies/Berachain/Berapaw/interfaces/IBexVault.sol src/strategies/Berachain/Berapaw/interfaces/IStrategy.sol src/strategies/Berachain/Berapaw/interfaces/IBalancerQuery.sol src/strategies/Berachain/Berapaw/interfaces/IRewardVaul.sol src/strategies/Berachain/Berapaw/BerapawStrategy.sol src/strategies/Berachain/Infrared/InfraredStrategy.sol src/strategies/Berachain/Infrared/InfraredStableStrategy.sol src/interfaces/IRagaVault.sol src/interfaces/IRagaVaultRegistry.sol src/interfaces/IRouter.sol
Commit Hash	7bd995827d954db0be7c3db6939a0f492e47f7c7
Language	Solidity
Blockchain	Berachain
Method	Manual Analysis, Functional Testing, Automated Testing

Review 1	9th June 2025 - 15th June 2025
Updated Code Received	18th June 2025
Review 2	18th June 2025 - 19th June 2025
Fixed In	https://github.com/Nexus-2023/NativeYield/commit/79f7f1459fa2113bf00c355680f3e4b0fc42852b

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	1 (5.9%)
🟥	High	5 (29.4%)
🟧	Medium	5 (29.4%)
🟨	Low	4 (23.5%)
🟪	Informational	2 (11.8%)

Severity

Issues

	🟥 Critical	🟥 High	🟧 Medium	🟨 Low	🟪 Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	1	0
Partially Resolved	0	0	0	0	0
Resolved	0	0	0	2	4

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Incorrect Staking of Accumulated Fees in Deposit	Critical	Resolved
2	User Funds Become Permanently Inaccessible After emergencyWithdraw Due to Missing Idle Balance Redemption Logic	High	Resolved
3	while calling withdrawAll() , all fees accumulated also being transferred to vault, leading to 0 fees paid to treasury	High	Resolved
4	totalAssets value in RagaVault will inflated due to the Accumulated fees on Strategies	High	Resolved
5	Missing Access Control in setVault Function	High	Acknowledged
6	Incorrect Logic in Panic Function.	High	Resolved
7	Incorrect JoinKind used in _buildHoneyByUsdLp -> _joinReq – single-token join bug	Medium	Resolved
8	Incorrect Swap Formula in InfraredStrategy	Medium	Resolved

Issue No.	Issue Title	Severity	Status
9	Dimension Mismatch of amounts array in InfraredStableStrategy::_joinReq Function	Medium	Resolved
10	Attacker can make the RagaVault::mint function unusable with Dust amounts	Medium	Acknowledged
11	Fee Validation Logic Error.	Medium	Resolved
12	Critical Griefing Attack via Share Price Inflation due to Missing _decimalsOffset() Override in ERC4626Upgradeable Vault	Low	Acknowledged
13	usage of slot0, for price fetch and its usage is prone to price manipulation	Low	Acknowledged
14	Incorrect Function Visibility in InfraredStableStrategy	Low	Resolved
15	Missing Zero Address Validation for Admin	Low	Resolved
16	Unsafe Direct approve Usage in BerapawStrategy Causes Potential Staking Failures and Incompatibility with Some ERC–20	Informational	Resolved
17	Missing nonReentrant modifier in withdrawAll function	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
- Implementation of ERC 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 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		
		🟥 High	🟧 Medium	🟨 Low
Likelihood	🟥 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.

Critical Severity Issues

Incorrect Staking of Accumulated Fees in Deposit

Resolved

Path

src/strategies/Berachain/Berapw/BerapwStrategy.sol

Function Name

deposit(), _harvestWithoutStake()

Description

In deposit all want tokens in address(this) are being staked in reward vault and s.totalAssets is incremented. And in _harvestWithoutStake() , gross - fee is only incremented , means fees is not incremented and not staked in reward vault.

but here issue lies that, when again deposit is being called, all this accumulated fee in strategy would be again be staked, and incremented in s.totalAssets, this leads to tresury would not be able to get all collected fees.

```
function deposit() external onlyVault nonReentrant {
       BerapawStrategyStorage storage s = _s();

       uint256 amt = want().balanceOf(address(this));
       s.totalAssets += amt;
       _harvest();
       want().approve(s.rewardVault, amt);
       IRewardVault(s.rewardVault).stake(amt);
    }
```

POC

1. vault deposit want() tokens through strategy.
2. s.totalAssets incremented
3. during the call, if there would have want() added in strategy during harvest, in that fees is calculated and that is not incremented in s.totalassetes
4. Now again when deposit is called , all balance is staked. means no tokens left.
5. In calculate fees , accumulated fees is incremented, and that is used to transferred
6. so after that everytime it will revert and so there is loss of funds for treasury.

Remediation

accumulated fees variable should be subtracted from total balance while depositing.

High Severity Issues

User Funds Become Permanently Inaccessible After emergencyWithdraw Due to Missing Idle Balance Redemption Logic

Resolved

Path

contracts/RagaVault.sol

Function

emergencyWithdraw(), _withdraw()

Description

The emergencyWithdraw() function pulls all underlying assets from the strategy into the vault. However, the _withdraw() function does not allow redemptions from the vault's local balance, instead relying solely on calling strategy.withdraw(assets) and measuring the change in local balance.

After an emergencyWithdraw(), the strategy holds no assets, and all tokens sit in the vault. When users later try to withdraw or redeem, the strategy call returns no funds, and the local balance delta is zero, so users receive nothing despite the vault holding the assets.

This creates a state where:

The vault has idle tokens.
Users have outstanding shares.
But no path exists for users to redeem those shares for tokens.
This results in permanently stuck funds and a complete denial of service to users.

```
function emergencyWithdraw() public onlyRegistryOwner whenPaused {
    IStrategy(_getVaultStorage().strategy).withdrawAll(); // moves assets to
vault
}


function _withdraw(
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
) internal override {
    …
    uint256 beforeBal = IERC20(asset()).balanceOf(address(this));
    IStrategy(_getVaultStorage().strategy).withdraw(assets);
    uint256 afterBal = IERC20(asset()).balanceOf(address(this));
    SafeERC20.safeTransfer(IERC20(asset()), receiver, afterBal - beforeBal);
}
```

After emergencyWithdraw, the call to strategy.withdraw(assets) does nothing.
afterBal - beforeBal = 0, so no tokens are transferred to the user.

POC

1. Normal operations ongoing, strategy manages 1,000 tokens.
2. Owner calls emergencyWithdraw() → vault now holds 1,000 tokens, strategy holds 0.
3. User A tries to withdraw or redeem 100 shares.
4. Vault calls strategy.withdraw(100) — strategy does nothing.
5. Balance difference is zero → user gets 0.
6. The 1,000 tokens remains permanently stuck in the vault.
7. No method is provided to recover them based on user share supply.

Remediation

Provide an admin function to recover idle tokens and allow manual refunds.

while calling withdrawAll() , all fees accumulated also being transferred to vault, leading to 0 fees paid to treasury

Resolved

Path

BeraPwStrategy.sol

Function

WithdrawAll()

Description

when calling emergency withdraw, all funds withdrawAll is called from strategy , this would lead to withdrawing of all want tokens from strategy, so in this all fees which are accumulated also, being removed from strategy, this will lead to treasury getting zero want tokens.

```
function emergencyWithdraw() public onlyRegistryOwner whenPaused {
        IStrategy(_getVaultStorage().strategy).withdrawAll();
    }

        uint256 totalOut = want().balanceOf(address(this));
        s.totalAssets = 0;
        want().safeTransfer(bs.vault, totalOut);
```

POC

1. Deposit called in strategy
2. harvest is called, fees is accumluated
3. WithdrawAll is called , strategy balance is now 0

Remediation

while withdrawing all funds, accumulated fees should be subtracted from balance of strategy.

totalAssets value in RagaVault will inflated due to the Accumulated fees on Strategies

Resolved

Path

src/strategies/Berachain/BaseStrategy.sol

Function

setVault

Description

totalAssets value in RagaVault will be inflated because of the fee accrued in strategies. If the fee accrued in strategies is unclaimed until the next deposit, that fee will be added into totalAssets (Which should not be the case because the fee amount will be claimed further) . This keeps on going for every deposit, which makes the value of totalAssets to be continuously inflated. This can also cause serious problems for Dos of Last withdrawers, as the early withdrawers get backs the inflated amounts.

Impact

- Inflation of totalAssets to a significant value
- Dos to Last withdrawers as the virtually inflated assets are not real , as fee no longer stays neither in strategies or in vault. And the initial withdrawer's got an extra portion due to the virtual inflation. Which finally affects Last withdrawers

```
  function totalAssets() public view override(ERC4626Upgradeable) returns
(uint256) {
        return IERC20(asset()).balanceOf(address(this)) +
IStrategy(_getVaultStorage().strategy).getTotalAssets();
    }
```

amt also includes the fee accrued.

```
 function deposit() external onlyVault nonReentrant {
        BerapawStrategyStorage storage s = _s();

        uint256 amt = want().balanceOf(address(this));
   @> s.totalAssets += amt;
        _harvest();
        want().approve(s.rewardVault, amt);
        IRewardVault(s.rewardVault).stake(amt);
    }
```

Remediation

- transfer the fee to another contract or address
- Or implement a separate fee mechanism.

Missing Access Control in setVault Function

Acknowledged

Path

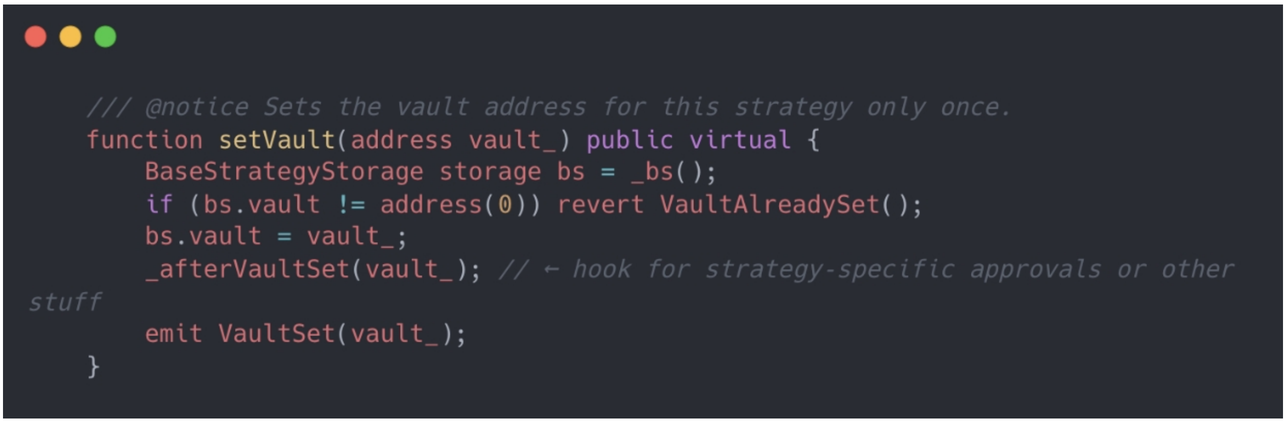
src/strategies/Berachain/BaseStrategy.sol

Function

setVault

Description

The setVault() function in BaseStrategyUpgradeable lacks proper access control. While the function is marked as virtual, none of the concrete strategy implementations override it with proper access control. This means anyone can call this function and set the vault address for any strategy contract.



```
    /// @notice Sets the vault address for this strategy only once.
    function setVault(address vault_) public virtual {
        BaseStrategyStorage storage bs = _bs();
        if (bs.vault != address(0)) revert VaultAlreadySet();
        bs.vault = vault_;
        _afterVaultSet(vault_); // ← hook for strategy-specific approvals or other
stuff
        emit VaultSet(vault_);
    }
```

Impact

An attacker can take control of strategy contracts by setting a malicious vault address.

Likelihood

The likelihood is high because the function is publicly accessible without any access control. Any external actor can call this function on deployed strategy contracts. The only protection is that the vault can only be set once, but if an attacker calls this function before the legitimate vault, they gain permanent control.

Recommendation

Add proper access control.

Incorrect Logic in Panic Function.

Resolved

Path

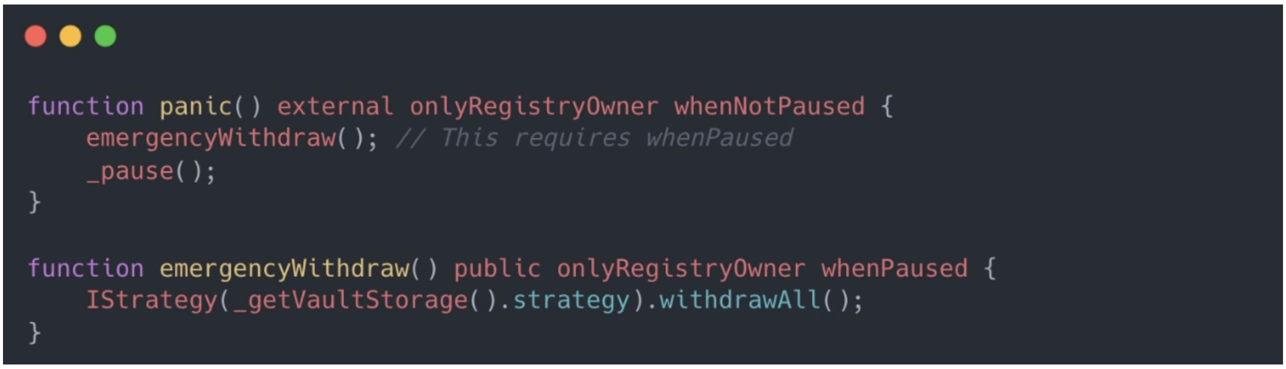
src/vaults/RagaVault.sol

Function

panic

Description

The panic() function contains a logical contradiction that makes it impossible to execute successfully. The function has a whenNotPaused modifier but calls emergencyWithdraw() which has a whenPaused modifier.



```
function panic() external onlyRegistryOwner whenNotPaused {
    emergencyWithdraw(); // This requires whenPaused
    _pause();
}

function emergencyWithdraw() public onlyRegistryOwner whenPaused {
    IStrategy(_getVaultStorage().strategy).withdrawAll();
}
```

Impact

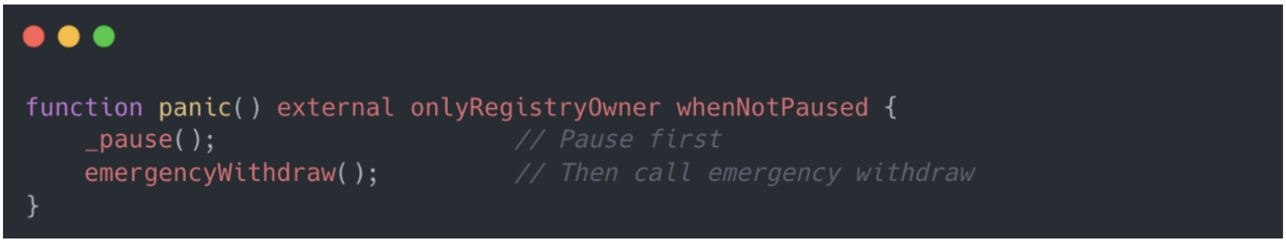
The panic function cannot be used.

Likelihood

The likelihood is high because every attempt to call the panic function will fail due to the modifier.

Recommendation

Change the order of execution in panic() function.



```
function panic() external onlyRegistryOwner whenNotPaused {
    _pause();                  // Pause first
    emergencyWithdraw();       // Then call emergency withdraw
}
```

Medium Severity Issues

Incorrect JoinKind used in _buildHoneyByUsdLp -> _joinReq – single-token join bug

Resolved

Path

InfraredStableStrategy.sol

Function

_buildHoneyByUsdLp | _joinReq

Description

The _buildHoneyByUsdLp -> _joinReq function performs a single-token join (only Honey), but it incorrectly uses EXACT_TOKENS_IN_FOR_BPT_OUT (JoinKind = 1). Per Balancer docs, single-token joins require TOKEN_IN_FOR_EXACT_BPT_OUT (JoinKind = 2). Using the wrong JoinKind leads to either revert on ZeroBPTReceived or no tokens received despite input being spent docs balancer.

Impact

High – Single-token deposits (Honey only) may be rejected or result in zero BPT minted, causing funds to be locked or lost.

Likelihood

Medium – Happens consistently for any single-token join via this method.

```
function _joinReq(
    StableStorage storage ss,
    uint256 honeyAmount,
    uint256 minBpt // 0 for query-only calls
)
    private
    view
    returns (IBexVault.JoinPoolRequest memory)
{
    address[] memory assets = new address[](3);
    uint256[] memory maxAmt = new uint256[](3);
    uint256[] memory amounts = new uint256[](2);
    assets[0] = ss.byusd;
    assets[1] = ss.lpToken;
    assets[2] = ss.honey;
    maxAmt[2] = honeyAmount;
    amounts[1] = honeyAmount;
-        bytes memory qUserData = abi.encode(1, amounts, minBpt); //
EXACT_TOKENS_IN_FOR_BPT_OUT
        IBexVault.JoinPoolRequest memory qReq = IBexVault.JoinPoolRequest({
            assets: assets,
            maxAmountsIn: maxAmt,
            userData: qUserData,
            fromInternalBalance: false
        });

        return qReq;
    }
```

With Honey being the only token, this mis-specification causes inconsistent behavior and protocol deviations.

POC

1. Call _buildHoneyByUsdLp passing honeyAmt > 0.
2. Balancer's queryJoin returns expBpt > 0, but since JoinKind=1 with only one token provided, pool rejects and returns 0 or reverts.
3. As a result, the function reverts with ZeroBPTReceived().

Alternately, the join may succeed but mint zero BPT, effectively burning Honey.

Remediation

- Use JoinKind = TOKEN_IN_FOR_EXACT_BPT_OUT (value = 2) when depositing a single token.
- Construct userData using:

```
bytes memory qUserData = abi.encode(2, amounts, minBpt); //
TOKEN_IN_FOR_EXACT_BPT_OUT
```

Incorrect Swap Formula in InfraredStrategy

Resolved

Path

InfraredStrategy.sol

Function

_calculateSwapAmount()

Description

There is a mathematical error in the _calculateSwapAmount function when calculating the amount of iBGT to swap in the iBGT-heavy case. The current implementation uses an incorrect formula that results in suboptimal LP token creation, reducing user yield.

```
// Incorrect implementation:
if (rightTerm > leftTerm) {
    // Case B: iBGT-heavy
    uint256 numerator = rightTerm - leftTerm;
    uint256 denominator = R + price;
    ibgtToSwap = numerator / denominator;
}
```

The current formula incorrectly adds values with different units:

- R is the wBERA:iBGT ratio (scaled by 1e18)
- price is the iBGT:wBERA ratio (scaled by 1e18)

When swapping iBGT for wBERA, we should get:
wberaReceived = ibgtToSwap * 1e18 / price

For optimal ratio after swap:
(wberaBal + ibgtToSwap * 1e18 / price) * 1e18 = R * (ibgtBal - ibgtToSwap)

Solving for ibgtToSwap gives the correct formula:
ibgtToSwap = (R * ibgtBal - wberaBal * 1e18) / (R + 1e36 / price)

POC

Given realistic values:

- wBERA balance = 5 (5 * 10^18)
- iBGT balance = 10 (10 * 10^18)
- Pool ratio (R) = 3 * 10^18 (3:1 wBERA:iBGT)
- Price = 2 * 10^18 (2 iBGT per wBERA)

Current formula calculates ibgtToSwap = 5, resulting in new balances of 7.5 wBERA and 5 iBGT (1.5:1 ratio).

Correct formula would calculate ibgtToSwap ≈ 7.14, resulting in balances of 8.57 wBERA and 2.86 iBGT (3:1 ratio).

Recommendation

Replace the current formula with the mathematically correct version:

```
if (rightTerm > leftTerm) {
    // Case B: iBGT-heavy
    uint256 numerator = rightTerm - leftTerm;
    uint256 denominator = R + Math.mulDiv(1e36, 1, price);
    ibgtToSwap = numerator / denominator;
}
```

Dimension Mismatch of amounts array in InfraredStableStrategy::_joinReq Function

Resolved

Path

InfraredStableStrategy.sol

Function

_joinReq

Description

In _joinReq, the assets array is initialized with length 3:

```
address;
// assets[0] = byUSD
// assets[1] = LP token
// assets[2] = HONEY
```

However, the amounts array is incorrectly initialized with length 2:

```
uint256[] memory amounts = new uint256[](2);
amounts[1] = honeyAmount;
```

Balancer's Vault API for EXACT_TOKENS_IN_FOR_BPT_OUT requires that the amounts (and maxAmountsIn) array length match the number of assets. A mismatch triggers a Dimension Mismatch revert when calling queryJoin.

Recommendation

Ensure that amounts is created with the same length as assets. For a single-sided HONEY join, you can set only the third element:

```
    function _joinReq(
        StableStorage storage ss,
        uint256 honeyAmount,
        uint256 minBpt // 0 for query-only calls
    )
        private
        view
        returns (IBexVault.JoinPoolRequest memory)
    {
        address[] memory assets = new address[](3);
        uint256[] memory maxAmt = new uint256[](3);
-       uint256[] memory amounts = new uint256[](2);
+      uint256[] memory amounts = new uint256[](3);
        assets[0] = ss.byusd;
        assets[1] = ss.lpToken;
        assets[2] = ss.honey;
        maxAmt[2] = honeyAmount;
-       amounts[1] = honeyAmount;
+      amounts[2] = honeyAmount;
```

```
        bytes memory qUserData = abi.encode(1, amounts, minBpt); //
EXACT_TOKENS_IN_FOR_BPT_OUT
        IBexVault.JoinPoolRequest memory qReq = IBexVault.JoinPoolRequest({
            assets: assets,
            maxAmountsIn: maxAmt,
            userData: qUserData,
            fromInternalBalance: false
        });

        return qReq;
    }
```

Note from Balancer Documentation

Balancer Vault's queryJoin for EXACT_TOKENS_IN_FOR_BPT_OUT expects userData to be encoded as:

1. kind (uint256)
2. amounts (uint256[]) with length equal to assets.length
3. minBPT (uint256)

In this contract, amounts.length must be 3 to match the three assets

Attacker can make the RagaVault::mint function unusable with Dust amounts

Acknowledged

Path

src/Vaults/RagaVault

Function

RagaVault::mint

Description

The attacker intends to exploit the vault's mint logic by sending a dust-level asset deposit (e.g., 1000 wei) directly to the vault when no shares exist. This leaves the vault in a state with non-zero assets but zero total shares. When an unsuspecting user later calls RagaVault::mint to mint shares, the vault calculates the required asset amount using an inflated value:

Attack Path

a. Attacker Initialization:
The attacker directly transfers a dust amount (e.g., 1000 wei) to the vault without minting any shares.
Vault state: assets = 1000 wei, totalShares = 0

b. Victim Deposit:
A normal user attempts to mint 10e18 shares via the RagaVault::mint() function.

c. Inflated Share Price Calculation:
Due to the vault having assets but no shares, the share-to-asset ratio becomes artificially inflated:

```
AssetsRequired = sharesToMint * (assets + 1) / (totalShares + 1)
= 10e18 * (1000 + 1) / (0 + 1)
= 10000e18 assets
```

The user is now required to deposit an unreasonably large amount of assets (10000x more) to mint the intended shares.

Outcome

- The share price is manipulated to an extremely high value using a negligible dust deposit.
- This may discourage or block all further mints, especially when high-value assets like WETH are used.
- It can effectively make RagaVault::mint() unusable.

```
function mint(uint256 shares, address receiver) public override whenNotPaused
returns (uint256 assets) {
        // Validation
        _depositPrechecks(shares, receiver);
        // Mint shares
        assets = super.mint(shares, receiver);
        // Move the funds to the strategy contract
        // note: check `totalAssets()` for doubts on withdrawal
        _pushToStrategy(assets);
    }
```

Recommendation

Instead of directly fetching the balance from balanceOf better use mappings for asset accounting.

Fee Validation Logic Error.

Resolved

Path

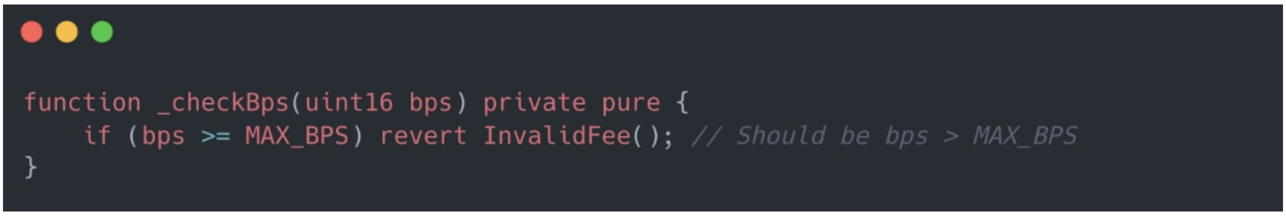
src/vaults/RagaVaultRegistry.sol

Function

_checkBps

Description

The fee validation function uses >= instead of > when checking against MAX_BPS, which prevents setting fees to exactly the maximum intended value of 5000 basis points (50%).



```
function _checkBps(uint16 bps) private pure {
    if (bps >= MAX_BPS) revert InvalidFee(); // Should be bps > MAX_BPS
}
```

Impact

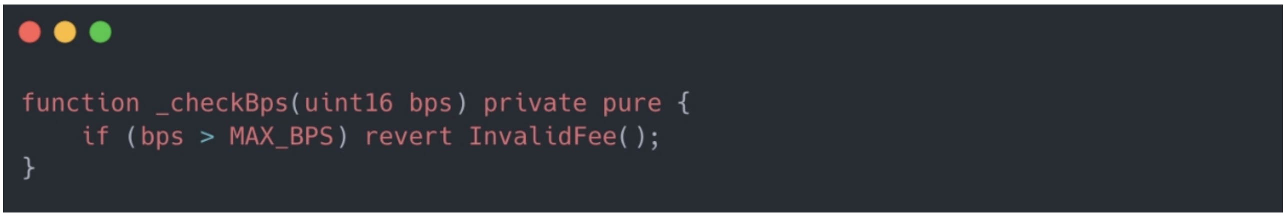
Prevents setting fees to the exact maximum value.

Likelihood

The likelihood is Medium because this only affects the edge case where someone wants to set fees to exactly 50%. Most fee settings would be well below this threshold

Recommendation

Change the comparison to allow setting fees to exactly MAX_BPS



```
function _checkBps(uint16 bps) private pure {
    if (bps > MAX_BPS) revert InvalidFee();
}
```

Low Severity Issues

Critical Griefing Attack via Share Price Inflation due to Missing _decimalsOffset() Override in ERC4626Upgradeable Vault

Acknowledged

Path

src/Vaults/RagaVault

Function

→ Inherits from ERC4626Upgradeable
→ Does not override _decimalsOffset()

Description

The RagaVault contract inherits from OpenZeppelin's ERC4626Upgradeable but fails to override the internal _decimalsOffset() function. By default, _decimalsOffset() returns 0.
This exposes the vault to a griefing attack during the first deposit:

- The first depositor mints X shares by depositing X assets.
- Then, they directly donate a large amount of the underlying asset to the vault.
- This inflates totalAssets() while totalSupply() remains X, massively increasing the price per share.
- Future depositors receive near-zero shares for normal-sized deposits.
- As a result, the vault becomes permanently unusable — a total denial of service.

This issue occurs due to the vault not overriding _decimalsOffset() to introduce an appropriate scaling factor. For ref - https://docs.openzeppelin.com/contracts/5.x/erc4626

```
// From ERC4626Upgradeable
function _decimalsOffset() internal view virtual returns (uint8) {
    return 0; // not overridden in RagaVault
}

function totalAssets() public view override returns (uint256) {
    return IERC20(asset()).balanceOf(address(this))
        + IStrategy(_getVaultStorage().strategy).getTotalAssets();
}
```

POC

1. Attacker calls deposit(1, attacker) → receives 1 share.
2. Attacker then calls token.transfer(vault, 1_000_000 e18) — donates funds.
3. totalAssets() now = (1,000,000 e18 + 1 wei). totalSupply() = 1.
4. Share price = 1,000,000 e18 + 1 wei token per share.
5. If a normal user tries to deposit any amount less than such big price, they get 0 shares.
6. Vault is effectively unusable and permanently griefed.

Remediation

Override _decimalsOffset() in the vault to set a proper share/asset scaling offset.

```
function _decimalsOffset() internal view override returns (uint8) {
    return 18;
}
```

usage of slot0, for price fetch and its usage is prone to price manipulation

Acknowledged

Path

InfraredStableStrategy and InfraredStrategy.sol

Function

_getPoolPrice(),_calculateExpectedOutput() etc

Description

In both the contract , to fetch price slot0, is used and that fetched sqrtPriceX96 is further used for calculation in swaps and estimation.The sqrtPriceX96 is pulled from Uniswap.slot0, which is the most recent data point and can be manipulated easily via MEV bots and Flashloans with sandwich attacks, which can cause the loss of funds. This could lead to wrong calculations and loss of funds for the vault and end users.
So whenever if there is somewhat low liquidty , attack is more feasible, eventhough if liquidiity is not low , it not nullifies the issue.

lets take one example , price fetch from slot0 is used in _buildWberaIbgtLp() and that price is used to calcaulate swap amount and that ibgttoswap is used in _mintLPTokens() ,so attack lead to manipulation of minting to lptokens. eventhough gas consumption would be somewhat high, but by figuring out where it is needed, at that place twap should be used, like in example we have given above.

```
    function _getPoolPrice(IUniV3Pool pool) internal view returns (uint256
price) {
        (uint160 sqrtPriceX96,,,,,,) = pool.slot0();
        return Math.mulDiv(uint256(sqrtPriceX96) * uint256(sqrtPriceX96), 1e18,
1 << 192);
    }
```

Recommendation

Use the TWAP function instead of slot0 to get the value of sqrtPriceX96. TWAP is a pricing algorithm used to calculate the average price of an asset over a set period. It is calculated by summing prices at multiple points across a set period and then dividing this total by the total number of price points.

Incorrect Function Visibility in InfraredStableStrategy

Resolved

Path

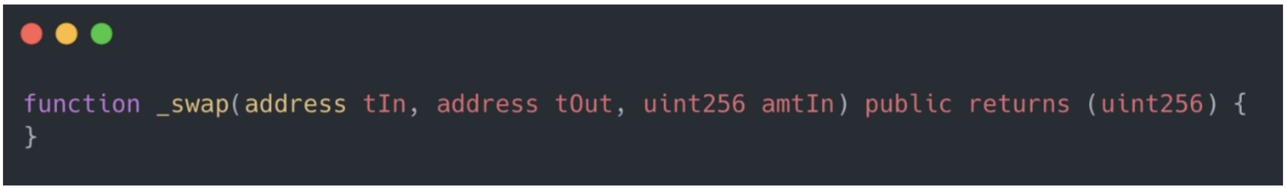
src/strategies/Berachain/Infrared/InfraredStableStrategy.sol

Function

_swap

Description

The _swap() function is marked as public instead of internal, which goes against the naming convention and intended design. Functions with underscore prefixes are typically meant to be internal



```
function _swap(address tIn, address tOut, uint256 amtIn) public returns (uint256) {
}
```

Impact

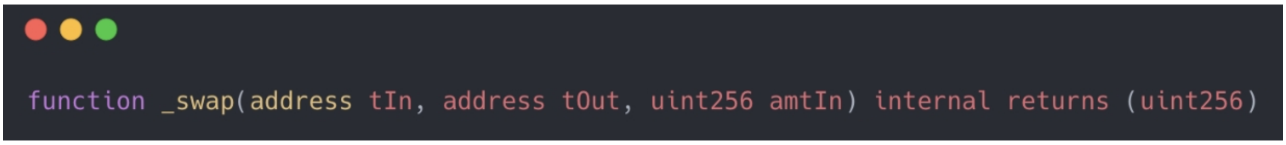
it breaks the intended design.

Likelihood

The likelihood is Low because the function is accessible and doesn’t have any impact on funds.

Recommendation

Change the function visibility from public to internal to match the intended design



```
function _swap(address tIn, address tOut, uint256 amtIn) internal returns (uint256)
```

Missing Zero Address Validation for Admin

Resolved

Path

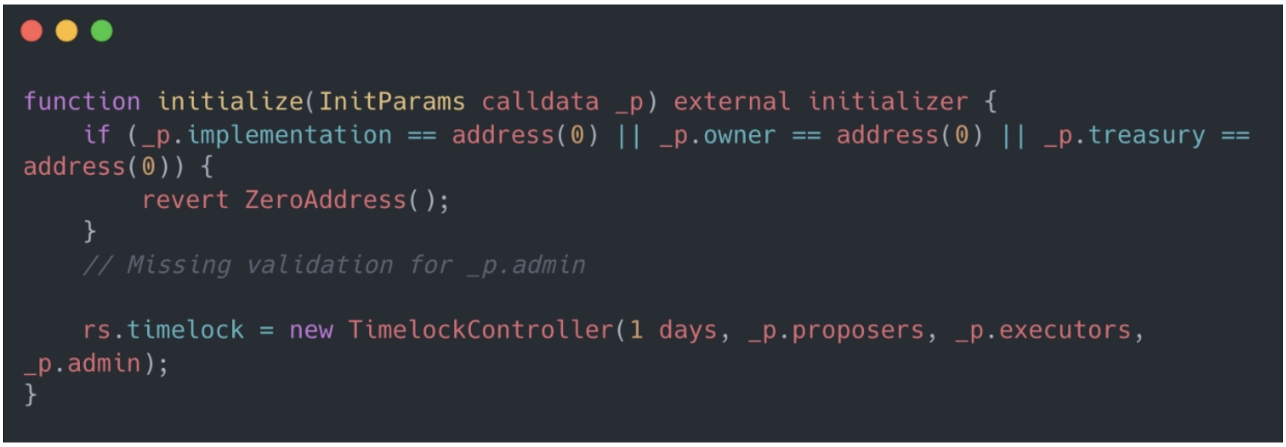
src/vaults/RagaVaultRegistry.sol

Function

initialize

Description

The initialize function validates that implementation, owner, and treasury addresses are not zero, but it doesn't validate the _p.admin address parameter.



```
function initialize(InitParams calldata _p) external initializer {
    if (_p.implementation == address(0) || _p.owner == address(0) || _p.treasury ==
address(0)) {
        revert ZeroAddress();
    }
    // Missing validation for _p.admin

    rs.timelock = new TimelockController(1 days, _p.proposers, _p.executors,
_p.admin);
}
```

Impact

Low

Likelihood

The likelihood is Low because it requires someone to accidentally pass a zero address for the admin parameter.

Recommendation

Add zero address validation for the admin parameter.

Informational Issues

Unsafe Direct approve Usage in BerapawStrategy Causes Potential Staking Failures and Incompatibility with Some ERC–20

Resolved

Path

BerapawStrategy.sol

Function

deposit(), internal _harvest(), internal _singleSidedJoin()

Description

The contract calls IERC20.approve(...) directly instead of using OpenZeppelin's SafeERC20.safeApprove(...). Some ERC–20 tokens (e.g., USDT, older or malicious tokens) revert if you try to change an existing non–zero allowance, or simply don't return a boolean. This can cause deposits, harvests, or single–sided joins to fail unexpectedly, breaking the strategy flow or leaving assets in limbo.

```
function deposit() external onlyVault nonReentrant {
    …
    // ❌ Direct approve without safety checks
    want().approve(s.rewardVault, amt);
    IRewardVault(s.rewardVault).stake(amt);
}

function _harvest() internal {
    …
    // ❌ Direct approve without safety checks
    want().approve(s.rewardVault, net);
    IRewardVault(s.rewardVault).stake(net);
}

function _singleSidedJoin(BerapawStrategyStorage storage s, uint256 amount)
internal {
    …
    // ❌ Direct approve without safety checks
    IERC20(s.lbgt).approve(s.bexVault, amount);
    …
}
```

Recommendation

Replace all approve(...) calls with OpenZeppelin's safe method

Missing nonReentrant modifier in withdrawAll function

Resolved

Path

/src/strategies/Berachain/Berapaw/BerapawStrategy.sol

Function

withdrawAll

Description

Missing nonReentrant in withdrawAll function in BerapawStrategy.

Recommendation

Add modifier

Functional Tests

Some of the tests performed are mentioned below:

- ✓ All state variables are correctly updated during deposit
- ✓ Access control verification
- ✓ Do fee calculations work correctly
- ✓ State variables updated correctly in emergency withdraw
- ✓ Initial state should be unpaused.

Threat Model

Contract	Function	Threats
RagaVaultRegistry	initialize	Zero address validation bypass, malicious admin/ proposer/executor setup, timelock misconfiguration
	setDepositFee	Fee manipulation, user fund extraction through excessive fees
	setManagementFee	Fee manipulation, user fund extraction through excessive fees
	setPerformanceFee	Fee manipulation, user fund extraction through excessive fees
	setWithdrawFee	Fee manipulation, user fund extraction through excessive fees
	deployVault	Malicious vault deployment, unauthorized vault creation
RagaVault	deposit	share price manipulation, front-running,inflation attacks
	mint	Share price manipulation, inflation attacks
	withdraw	Withdrawal blocking, fund lock-in, share price manipulation during exit

Contract	Function	Threats
BaseStrategy	redeem	Withdrawal blocking, fund lock-in, share price manipulation during exit
	totalAssets	Incorrect asset reporting, share price manipulation, accounting fraud
	pause	Unauthorized pausing, DoS attacks, fund lock-in, emergency abuse
	unpause	Unauthorized unpausing, premature resumption during emergencies
	emergencyWithdraw	Logic contradiction preventing execution
	setVault	Unauthorized vault setting, strategy hijacking, complete fund theft
	setSlippageBps	Slippage manipulation, sandwich attacks
	collectFees	Unauthorized fee collection, fund drainage, treasury manipulation
	want	incorrect asset targeting
	getAccumulatedFee	Fee accounting manipulation, incorrect fee calculations
InfraredStableStrategy	initialize	Parameter manipulation, malicious configuration, protocol integration attacks

Contract	Function	Threats
	deposit	Reentrancy attacks, fund misallocation, strategy manipulation
	withdraw	User receiving more than requested , protocol draining
	withdrawAll	Emergency withdrawal failure, fund lock-in during crisis
	harvest	reward manipulation, fee calculation errors
	getTotalAssets	Incorrect asset calculation, share price manipulation
	tvlWithReward	TVL manipulation, reward calculation
	_swap	Unauthorized external access, slippage exploitation, calculation errors.
InfraredStrategy	initialize	malicious configuration, protocol integration attacks
	deposit	fund misallocation, strategy manipulation
	withdraw	Fund over-distribution, user receiving more than requested, protocol drainage
	tvlWithReward	TVL manipulation, reward calculation errors
BerapawStrategy	deposit	Reentrancy attacks, fund misallocation, strategy manipulation

Contract	Function	Threats
	withdraw	Insufficient asset validation, withdrawal failures
	withdrawAll	Missing reentrancy protection
	getTotalAssets	Asset tracking errors, share price manipulation, accounting issues

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.

Closing Summary

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

Issues of Critical, High, Medium, Low and Informational severity were found. Raga Finance team acknowledged a few and resolved the remaining issues.

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 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.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

June 2025

For

raga

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com