SMART CONTRACTS REVIEW

zokyo

January 8th 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING YIELDNEST

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 0 High issues: 0 points deducted
- 2 Medium issues: 1 acknowledged and 1 partially resolved = - 5 points deducted
- 3 Low issues: 2 acknowledged and 1 resolved = - 2 points deducted
- 3 Informational issues: 1 resolved and 2 acknowledged = 0 points deducted

Thus, 100 - 5 - 2 = 93

TECHNICAL SUMMARY

This document outlines the overall security of the Yieldnest smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Yieldnest smart/s contract codebase for quality, security, and correctness.

Contract Status

There were 0 critical issue found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Yieldnest team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Yieldnest repository:

Repo: https://github.com/yieldnest/yieldnest-kernel-lrt/tree/feat/ynBTCk-strategy
Last fix - https://github.com/yieldnest/yieldnest-kernel-lrt/pull/22/files

Contracts under the scope:

- src/module/BaseKernelRateProvider.sol
- src/module/KernelRateProvider.sol
- src/module/BTCRateProvider.sol
- src/KernelStrategy.sol
- src/KernelClisStrategy.sol
- src/MigratedKernelStrategy.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Yieldnest smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The smart contracts form a modular system for managing and interacting with yield-generating strategies and tokenized assets in a DeFi ecosystem. `KernelStrategy` serves as the base contract, defining the framework for strategy implementation, including deposit and withdrawal mechanisms. `KernelClisStrategy` and `MigratedKernelStrategy` extend this base, introducing specific functionality like synchronized deposit/withdrawal handling, asset migration, and integration with external staking gateways such as `IStakerGateway`. The `BNBRateProvider` and `BTCRateProvider` contracts implement rate fetching for assets like WBNB, BNBX, and BTCB, facilitating accurate asset valuation through interactions with external managers like `IBNBXStakeManagerV2`. Finally, `BaseKernelRateProvider` provides foundational logic for handling kernel vault assets and rate derivations, allowing extensions like the rate providers to seamlessly integrate with staker gateways and kernel vaults. Together, these contracts enable efficient asset management, staking, and strategy execution while maintaining extensibility for additional assets and protocols.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Yieldnest team and the Yieldnest team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	hasAllocators is not Initialized	Medium	Acknowledged
2	Potential for Locked Funds in _withdrawAsset Function	Medium	Partially Resolved
3	Ambiguous Return Value in tryGetVaultAsset Function	Low	Resolved
4	Potential Infinite Recursion in getRate Function	Low	Acknowledged
5	Assumption of 1:1 Ratio and Comment Inconsistency in BNBRateProvider and BTCRateProvider Contracts	Low	Acknowledged
6	Unresolved TODO for referralId	Informational	Acknowledged
7	The parameter description is missing	Informational	Resolved
8	Floating pragma	Informational	Acknowledged

`hasAllocators` is not Initialized

The `onlyAllocator` modifier checks the `hasAllocators` flag to enforce role-based access. If `hasAllocators` is `false`, the check for the `ALLOCATOR_ROLE` is bypassed, allowing unauthorized users to perform restricted actions. Since `hasAllocators` is not initialized to `true`, this issue can lead to unauthorized access by default.

File: KernelStrategy.sol

```
391:     modifier onlyAllocator() {
392:         if (_getStrategyStorage().hasAllocators && !hasRole(ALLOCATOR_ROLE,
msg.sender)) {
393:             revert AccessControlUnauthorizedAccount(msg.sender, ALLOCATOR_ROLE);
394:         }
395:         _;
396:     }
```

Recommendation:

Initialize the `hasAllocators` variable to `true` in the `initialize` function to enforce stricter access control.

Client comment: Acknowledged. Will fix. In theory, one can set hasAllocators at vault creation time, but this value set to true is a safer default.

Potential for Locked Funds in _withdrawAsset Function

The _withdrawAsset function in the KernelClisStrategy contract presents a significant risk of locking user funds under certain conditions. The relevant code segment is as follows:

Current implementation:

```
if (vaultBalance < assets && strategyStorage.syncWithdraw) {
    // TODO: fix referralId
    string memory referralId = "";
    IStakerGateway(strategyStorage.stakerGateway).unstakeClisBNB(assets,
referralId);

    //wrap native token
    IWBNB(asset()).deposit{value: assets}();
}

SafeERC20.safeTransfer(IERC20(asset_), receiver, assets);
```

Issues Identified:

2. Conditional Execution of Unstake and Wrap:
    - If syncWithdraw is false and vaultBalance < assets, the function does not attempt to unstake CLIS BNB or wrap native BNB. This means that if the contract's balance is insufficient to cover the withdrawal request, the withdrawal will fail without any attempt to rectify the situation.
3. Safe Transfer Without Sufficient Balance Check:
    - The safeTransfer call is executed regardless of whether the unstaking and wrapping operations were performed or successful. This can lead to a situation where the transfer fails due to insufficient balance, causing the entire transaction to revert.

Potential Scenarios:
A. If syncWithdraw is false and vaultBalance < assets, the safeTransfer call will fail, resulting in a revert of the entire transaction.
B. Even if syncWithdraw is true, there are no checks to ensure that the unstaking and wrapping operations were successful before attempting to transfer assets.

Recommendation:

To address these issues, consider implementing the following changes:

1. Ensure Sufficient Balance Before Transfer:
   - Add a check to confirm that there is sufficient balance before executing the safeTransfer, regardless of the syncWithdraw status.
2. Handle Failures in Unstaking and Wrapping Operations:
   - Implement proper error handling for both unstaking and wrapping operations to ensure that any failures are addressed before proceeding with asset transfers.
3. Allow Partial Withdrawals:
   - Consider modifying the logic to allow for partial withdrawals if full withdrawal cannot be fulfilled due to insufficient balance.

```
function _withdrawAsset(
    address asset_,
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
) internal override onlyRole(ALLOCATOR_ROLE) {
    VaultStorage storage vaultStorage = _getVaultStorage();
    StrategyStorage storage strategyStorage = _getStrategyStorage();

    uint256 vaultBalance = IERC20(asset_).balanceOf(address(this));

    if (vaultBalance < assets) {
        if (strategyStorage.syncWithdraw) {
            // Attempt to unstake and wrap
            try
IStakerGateway(strategyStorage.stakerGateway).unstakeClisBNB(assets, "") {
                try IWBNB(asset()).deposit{value: assets}() {
                    // Unstaking and wrapping successful
                } catch {
                    revert("Wrapping failed");
                }
            } catch {
                revert("Unstaking failed");
            }
        } else {
            revert("Insufficient balance and syncWithdraw is false");
        }
    }

    // Recheck balance after potential unstaking and wrapping
    vaultBalance = IERC20(asset_).balanceOf(address(this));
    require(vaultBalance >= assets, "Insufficient balance after
unstaking");
```

```
    vaultStorage.totalAssets -= assets;
    if (caller != owner) {
        _spendAllowance(owner, caller, shares);
    }

    SafeERC20.safeTransfer(IERC20(asset_), receiver, assets);
    _burn(owner, shares);

    emit WithdrawAsset(caller, receiver, owner, asset_, assets, shares);
}
```

Client comment:

- If syncWithdraw is false and vaultBalance < assets, the function does not attempt to unstake CLIS BNB or wrap native BNB. This means that if the contract's balance is insufficient to cover the withdrawal request, the withdrawal will fail without any attempt to rectify the situation." - This is by design. Normally syncWithdraw is true.
- If syncWithdraw is false and vaultBalance < assets, the safeTransfer call will fail, resulting in a revert of the entire transaction. - This is by design.
- When syncWithdraw is set to false, the mode of operation is the following:
- ALLOCATOR_MANAGER_ROLE calls BaseVault.processor and calls unstakeClisBNB for amounts that then sit idle in the protocol and users withdraw them.
- Thus the contents of the vault act as a buffer and users can withdraw only when buffer has contents.

What changes we did decide to make:
https://github.com/yieldnest/yieldnest-kernel-lrt/blob/fe74305e491d49c64285129d64e7cdd6a5c4b3db/src/KernelStrategy.sol#L339
Unstake only assets - vaultBalance
This means the user needs is unstaked, no more.
Unstaking the full assets as before, means that the vault may not allow the user to withdraw asset A even if overall there's enough of asset A, if some of it is existing unstaked, which is an issue.
Define a _stake and _unstake handler to simplify the code and deduplicate the code in KernelClisStrategy.sol
https://github.com/yieldnest/yieldnest-kernel-lrt/blob/fe74305e491d49c64285129d64e7cdd6a5c4b3db/src/KernelStrategy.sol#L271

Changes can be seen in the closed pull Request below:
https://github.com/yieldnest/yieldnest-kernel-lrt/pull/32/files
<pending>

Ambiguous Return Value in tryGetVaultAsset Function

The tryGetVaultAsset function in the BaseKernelRateProvider contract returns address(0) in two distinct scenarios:

1. When the getVault check fails (IStakerGateway(MC.STAKER_GATEWAY).getVault(asset) != vault)
2. When the external call to IKernelVault(vault).getAsset() throws an exception

This ambiguity in the return value makes it difficult to distinguish between these two failure modes, potentially leading to incorrect error handling or decision-making in contracts that rely on this function.

Current implementation:

```
function tryGetVaultAsset(address vault) public view returns (address) {
    try IKernelVault(vault).getAsset() returns (address asset) {
        if (IStakerGateway(MC.STAKER_GATEWAY).getVault(asset) != vault) {
            return address(0);
        }
        return asset;
    } catch {
        return address(0);
    }
}
```

Recommendation:

Implement custom errors to provide more specific and gas-efficient error handling. This approach allows callers to distinguish between different failure scenarios while maintaining the original return type for successful cases.

```
error AssetMismatch(address vault, address asset);
error ExternalCallFailed(address vault);

function tryGetVaultAsset(address vault) public view returns (address) {
    try IKernelVault(vault).getAsset() returns (address asset) {
        if (IStakerGateway(MC.STAKER_GATEWAY).getVault(asset) != vault) {
            revert AssetMismatch(vault, asset);
        }
        return asset;
    } catch {
        revert ExternalCallFailed(vault);
    }
}
```

Client comment:

Acknowledged. Partial fix: We're only adding: revert AssetMismatch(vault, asset);
We want to be able to run code in the getRate() function after this call and reverting on ExternalCallFailed and try catching again is pedantic.

Potential Infinite Recursion in getRate Function

In both the BNBRateProvider and BTCRateProvider contracts, there is a potential for infinite recursion in the getRate function. This issue arises when tryGetVaultAsset returns a non-zero address that is not one of the explicitly handled assets. In such cases, the function calls itself recursively without any mechanism to break the recursion.
The problematic code section is:

```
address vaultAsset = tryGetVaultAsset(asset);

if (vaultAsset != address(0)) {
    return getRate(vaultAsset);
}
```

If there's a chain of vault assets referencing each other, this could lead to a stack overflow, causing the transaction to fail and potentially rendering the contract unusable for certain inputs.

Recommendation:
To prevent infinite recursion, implement a depth limit for recursive calls. This can be achieved by adding a parameter to track the recursion depth and reverting if it exceeds a predefined maximum. Here's a proposed modification:

```
function getRate(address asset) public view override returns (uint256) {
    return getRateWithDepth(asset, 0);
}

function getRateWithDepth(address asset, uint256 depth) private view
returns (uint256) {
    if (depth > 5) { // Set a reasonable maximum depth
        revert("Max recursion depth exceeded");
    }

    // Existing asset checks...

    address vaultAsset = tryGetVaultAsset(asset);

    if (vaultAsset != address(0)) {
        return getRateWithDepth(vaultAsset, depth + 1);
    }

    revert UnsupportedAsset(asset);
}
```

Client comment:

Security assumption: Kernel is assumed to not self-reference its own values in getAsset(). This assumption is on the same level as the assumption that getAsset() and getVault() do not revert, which the code already assumes in this implementation.
In either of these cases, the vault will be paused by the vault manager, and offending vaults will be removed.

Assumption of 1:1 Ratio and Comment Inconsistency in BNBRateProvider and BTCRateProvider Contracts

The BNBRateProvider and BTCRateProvider contracts make critical assumptions regarding asset valuation that could lead to inaccuracies in rate calculations:

1. Assumption of 1:1 Ratio:
    a. The contracts assume a 1:1 exchange rate for certain assets, specifically for CLISBNB in BNBRateProvider and all assets in BTCRateProvider. This assumption may not hold true under varying market conditions or if the underlying mechanisms for these assets change. If the actual exchange rates deviate from this assumption, it could result in incorrect rate calculations, impacting users who rely on these rates for transactions or further financial operations.
2. Comment Inconsistency:
    a. Both contracts contain comments stating, "add a multiplier to the rate if kernel changes from 1:1." However, there is no implementation of such a multiplier in the code. This inconsistency between the comments and the actual logic can mislead future developers or auditors regarding the intended functionality of the contracts. It may lead to confusion about whether additional logic is required or if the current implementation is sufficient.

Recommendation:

To address these issues, consider the following actions:

1. Implement Rate Multipliers:
    a. If there are scenarios where a multiplier should be applied to adjust for varying exchange rates, implement this logic in the getRate function. Ensure that any such multipliers are clearly defined and documented.
2. Update Comments for Clarity:
    a. Revise comments within the code to accurately reflect the implemented logic. If no multiplier is currently used, remove misleading comments or clarify that they are placeholders for future development.

Unresolved TODO for `referralId`

There are unresolved TODO comments in `_deposit` and `_withdrawAsset` functions for the `referralId` parameter when interacting with the `IStakerGateway`.

Recommendation:
Define and implement the logic for `referralId` in both `_deposit` and `_withdrawAsset` functions. If not needed, explicitly remove the TODO and associated parameter to avoid ambiguity.

Client comment: Acknowledged. Removed todo, Won't fix.
The contract bytecode size is too large, we're skipping this feature.

The parameter description is missing

Location: KernelStrategy.sol

The _withdrawAsset() function is only missing a description for the “asset_” parameter even if it has other parameter descriptions.

```
/**
     * @notice Internal function to handle withdrawals.
     * @param caller The address of the caller.
     * @param receiver The address of the receiver.
     * @param owner The address of the owner.
     * @param assets The amount of assets to withdraw.
     * @param shares The equivalent amount of shares.
     * @dev This is an example:
     *      The _withdraw function for strategies needs an override
     */
function _withdrawAsset(
    address asset_,
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
) internal virtual onlyAllocator {
```

Recommendation:
Add a description for the “asset_” parameter.

Floating pragma

The contracts use a floating pragma version `(^0.8.24)`. Contracts should be deployed using the same compiler version and settings as were used during development and testing. Locking the pragma version helps ensure that contracts are not inadvertently deployed with a different compiler version.

Recommendation:
Consider locking the pragma version to a specific, tested version to ensure consistent compilation and behavior of the smart contract.

Client comment: Acknowledged. Won't fix. We prefer the floating pragma.

	src/module/BaseKernelRateProvider.sol **src/module/KernelRateProvider.sol** **src/module/BTCRateProvider.sol** **src/KernelStrategy.sol** **src/KernelClisStrategy.sol** **src/MigratedKernelStrategy.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Yieldnest team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Yieldnest team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo