# YieldYak Audit

ERC 4626

June 2022

By CoinFabrik

# Introduction

CoinFabrik was asked to audit the contracts for the YieldYak project. First we will provide a summary of our discoveries, and then we will show the details of our findings.

## Scope

The audited files are from the git repository located at https://github.com/midnight-commit/smart-contracts. The audit is based on the commit `6a57e60e1eebadda288662c627c98a5f2ea9ffd7`.

The audited files are:

- `contracts/strategies/VariableRewardsStrategy.sol`: It contains the abstract `VariableRewardsStrategy` contract. It can be used to develop strategies where there are multiple reward tokens and the developer can change them after the contract is deployed. It is derived from the `YakStrategyV3` contract.
- `contracts/YakStrategyV3.sol`: It contains the `YakStrategyV3` abstract contract. This is the third iteration of the base contract used to make investment strategies, which is derived from the `YakBase` contract.
- `contracts/YakBase.sol`: It contains the `YakBase` abstract contract, which has the base functionality intended to be shared by vaults and strategies. It must be noted that the derived contracts are intended to conform to EIP-4626. It is derived from the `Ownable` contract.
- `contracts/lib/Ownable.sol`: It contains the `Ownable` abstract contract, having the utilities used in contracts with an owner.
- `contracts/lib/ERC20.sol`: It contains the base contract used to make ERC20 tokens.
- `contracts/lib/DexLibrary.sol`: It contains utilities used to interact with decentralized exchanges.
- `contracts/lib/SafeERC20.sol`: It contains utilities to safely interact with ERC20 contracts.
- `contracts/interfaces/IERC4626.sol`: It contains an interface definition for the ERC4626 standard.
- `contracts/interfaces/IPair.sol`: It contains an interface with the API definition for Uniswap-like pairs.
- `contracts/interfaces/IWAVAX.sol`: It contains an interface with the API definition for the WAVAX token.

The scope of the audit is limited to those files. No other files in this repository were audited. Its dependencies are assumed to work according to their documentation. Also, no tests were reviewed for this audit.

## Analyses

Without being limited to them, the audit process included the following analyses:

- Arithmetic errors
- Outdated version of Solidity compiler
- Race conditions
- Reentrancy attacks
- Misuse of block timestamps
- Denial of service attacks
- Excessive gas usage
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures
- Centralization and upgradeability

# Summary of Findings

We found no critical issues, 2 medium issues and several minor issues. Also, several enhancements were proposed.

## Security Issues

| ID | Title | Severity | Status |
|---|---|---|---|
| ME-01 | Reinvest Denial of Service | Medium | Unresolved |
| ME-02 | rewardToken must be WAVAX on VariableRewardsStrategy | Medium | Unresolved |
| MI-01 | Possible Reentrancy Issues in YakBase | Minor | Unresolved |
| MI-02 | Phantom Overflows When Multiply and Divide | Minor | Unresolved |

| ID | Title | Severity | Status |
|---|---|---|---|
| MI-03 | VariableRewardsStrategy._reinvest() Reentrancy Requirements | Minor | Unresolved |
| MI-04 | Issues Adding and Removing Reward Tokens in VariableRewardsStrategy | Minor | Unresolved |
| MI-05 | Deposit-Reinvest Coupling | Minor | Unresolved |

# Security Issues Found

## Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of, but can be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.

## Issues Status

An issue detected by this audit can have four distinct statuses:

- **Unresolved**: The issue has not been resolved.
- **Acknowledged**: The issue remains in the code, but is a result of an intentional decision.
- **Resolved**: Adjusted program implementation to eliminate the risk.
- **Partially resolved**: Adjusted program implementation to eliminate part of the risk. The other part remains in the code, but is a result of an intentional decision.
- **Mitigated**: Implemented actions to minimize the impact or likelihood of the risk

# Critical Severity Issues

No issues were found.

# Medium Severity Issues

## ME-01 Reinvest Denial of Service

**Location**:

- `contracts/YakStrategyV3.sol: 139-142`
- `contracts/strategies/VariableRewardsStrategy.sol: 185,190`

The reinvestment procedure defined in the `VariableRewardsStrategy._reinvest()` function can be blocked if any of the calls to transfer fees reverts. This can be trivially triggered by the dev via setting the dev to the address zero invoking the `updateDevAddr()` function. Reverting on transfer to the zero address is the normal behavior of ERC20 tokens.

### Recommendation

A zero-check can be added to the `YakStrategyV3.updateDevAddr()` function to mitigate this issue and not make it trivially exploitable. In order to fully solve this issue, use the pull-over-push pattern to transfer fees in the `VariableRewardsStrategy._reinvest()`. This solution will protect the strategies from any unexpected behavior in the `rewardToken` that may cause the transfer to revert, preventing this kind of denial of service attacks.

### Status

**Unresolved**.

## ME-02 rewardToken must be WAVAX on VariableRewardsStrategy

**Location**:

- `contracts/strategies/VariableRewardsStrategy.sol`

In several parts of the VariableRewardsStrategy it is assumed that the rewardToken is WAVAX. For example, see lines 176 and 199 of `contracts/strategies/VariableRewardsStrategy.sol`. If not, the behavior is not expected. But there is no guarantee that it is the case.

### Recommendation

Add a check in the `VariableRewardsStrategy` constructor, checking that `rewardToken` is WAVAX.

Status
**Unresolved**.

# Minor Severity Issues

## MI-01 Possible Reentrancy Issues in YakBase

**Location**:

- `contracts/YakBase.sol: 99-184`

Several functions in the YakBase contract follow the following structure:

```
require(checkOverridableInChildContract());
// some other things
actionNotDefinedInCurrentClass();
```

This leaves a possible reentrancy issue that will be hidden in the inheritance tree.

For example, see the deposit function in line 99 of `contracts/YakBase.sol` (underlined the relevant parts)

```
function deposit(uint256 _assets, address _receiver) public override returns (uint256
shares) {
   require(_assets <= maxDeposit(_receiver), "YakBase::Deposit more than max");

   shares = previewDeposit(_assets);
   IERC20(asset).safeTransferFrom(msg.sender, address(this), _assets);

   _mint(_receiver, shares);

   deposit(_assets, shares);

   emit Deposit(msg.sender, _receiver, _assets, shares);
}
```

Here a reentrancy attack may be able to circumvent the `maxDeposit()` check if the `maxDeposit()` and `deposit()` have been defined in a way to allow it in the child contract.

We found the following functions of the `YakBase` contract to have this problem:

- `deposit()`
- `mint()`
- `withdraw()`
- `redeem()`

### Recommendation
Make the functions with this issue non-reentrant.

### Status
**Unresolved**.

## MI-02 Phantom Overflows When Multiply and Divide

**Location**:

- `contracts/YakBase.sol: 203,217`
- `contracts/YakStrategyV3.sol: 184`
- `contracts/strategies/VariableRewardsStrategy.sol: 76,93,165,183,188`
- `contracts/lib/DexLibrary.sol: 130`

When multiplying and dividing 3 256-bit numbers (`a*b/c`), the intermediate result of multiplying `a*b` may exceed 256 bits even if the final result is under 256bits. This will trigger an overflow, reverting the transaction.

### Recommendation
In order to avoid the unwanted overflow, implement the multiply and divide operation to handle bigger numbers after the multiplication. If multiplying 256bit numbers, it should handle 512-bit results.

### Status
**Unresolved**.

### MI-03 VariableRewardsStrategy._reinvest() Reentrancy Requirements

**Location**:

- `contracts/strategies/VariableRewardsStrategy.sol: 120,176-197`
- `contracts/YakBase.sol: 107`

In the `VariableRewardsStrategy` contract, the `_reinvest()` function, defined in line 176 of `contracts/strategies/VariableRewardsStrategy.sol`, depends on the `_getRewards()`, `_convertRewardTokenToDepositToken()` and `_stakeDepositTokens()` functions, which are defined in the derived contracts. This makes it very difficult to ensure that a reentrancy attack will not be made when implementing those functions.

It must be noted that `_reinvest()` is called in the `deposit()` function, defined in line 116 of `contracts/strategies/VariableRewardsStrategy.sol`, which is called by `YakBase.deposit()` (defined in line 99 of `contracts/YakBase.sol`) and it is not guarded by a reentrancy guard.

#### Recommendation

a. Document that the `_reinvest()` function should only be used in reentrancy-guarded functions.
b. Either remove the call to `_reinvest()` in `VariableRewardsStrategy.deposit()` or make `YakBase.deposit()` non-reentrant.

#### Status

**Unresolved**.

### MI-04 Issues Adding and Removing Reward Tokens in VariableRewardsStrategy

**Location**:

- `contracts/strategies/VariableRewardsStrategy.sol: 29-31,248-278`

The `VariableRewardsStrategy._addReward()` and `VariableRewardsStrategy.removeReward()` functions have some issues, which derive from not using a standard implementation for the enumerable map.

These are the ones we found:

a. On `VariableRewardsStrategy._addReward()`, if `_rewardToken == rewardToken` it adds an uncontrolled swap-pair.

b. If the same `_rewardToken` is passed twice to `VariableRewardsStrategy._addReward()` the removal of the pair using `VariableRewardsStrategy.removeReward()` does not remove the reward.
c. `VariableRewardsStrategy._addReward()`, and `VariableStrategy.removeReward()` are O(n), but they can be O(1).

### Recommendation

a. Use OpenZeppelin's enumerable maps to implement the `reward->swapPair` mapping.
b. Do not use the swap-pair mapping to decide on AVAX to WAVAX conversion. Instead just do it every time or handle the decision separately.

### Status

**Unresolved**.

## MI-05 Deposit-Reinvest Coupling

**Location**:

- `contracts/strategies/VariableRewardsStrategy.sol: 117-122`

Sometimes a reinvest action is triggered by a deposit made by a user or another contract. See lines 116-124 of `contracts/strategies/VariableRewardsStrategy.sol`

```
function deposit(uint256 _assets, uint256) internal override {
   if (MAX_TOKENS_TO_DEPOSIT_WITHOUT_REINVEST > 0) {
       uint256 estimatedTotalReward = checkReward();
       if (estimatedTotalReward > MAX_TOKENS_TO_DEPOSIT_WITHOUT_REINVEST) {
           _reinvest(true);
       }
   }
   _stakeDepositTokens(_assets);
}
```

This coupling generates the following issues:

a. When the reinvestment is triggered on deposit, the `VariableRewardsStrategy.previewDeposit()` function may not comply with ERC 4626, which requires that this function "MUST return as close to and no more than the exact amount of Vault shares that would be minted in a deposit call in the same transaction"[1]. This may not be achieved because the `VariableRewardsStrategy.previewDeposit()` function does not take into account the gains that will be made via the reinvest process, which will probably reduce the amount of shares obtained by the depositor.
b. It allows a non-EOA address to trigger the reinvest action, which is not desired.
c. It makes the ME-01 and MI-03 issues worse.
d. When the reinvest is triggered on deposit, the depositor is forced to pay for an action that in the best case is not what they want to achieve and in the worst case makes it lose shares.

### Recommendation

Decouple deposit and reinvest actions by removing lines 117-122 of `contracts/strategies/VariableRewardsStrategy.sol`.

### Status

**Unresolved**.

[1] See https://eips.ethereum.org/EIPS/eip-4626.

# Enhancements

These items do not represent a security risk. They are best practices that we suggest implementing.

## Table

| ID | Title | Status |
|---|---|---|
| EN-01 | No Tests | Not implemented |
| EN-02 | Another Swap-Pairs Check In DexLibrary | Not implemented |

## Details

### EN-01 No Tests

There are no automated tests covering the functionality of the audited contracts in the repository.

#### Recommendation

Add tests covering the audited contracts to the repository and maintain them.

#### Status

**Not implemented**.

### EN-02 Another Swap-Pairs Check In DexLibrary

We noticed that the logic to validate that the `swapPairToken0` and `swapPairToken1` are correct in order to invoke the `DexLibrary.convertRewardTokensToDepositTokens()` function is repeated in a lot of your contracts.

#### Recommendation

Add an extra function in `DexLibrary` to check that `rewardToken` and `depositToken` are compatible with `swapPairToken0` and `swapPairToken1`. This new function would be similar in intent to `DexLibrary.checkSwapPairCompatibility()`, which checks that the pair is and addresses are correct to invoke `DexLibrary.swap()`.

#### Status

**Not implemented**.

# Other Considerations

The considerations stated in this section are not right or wrong. We do not suggest any action to fix them. But we consider that they may be of interest for other stakeholders of the project, including users of the audited contracts, owners or project investors.

## Centralization

The dev and the owner of each of the analyzed contracts have privileged capabilities.

The development team informed us that they are using a community multisig to help mitigate centralization risks.

## Upgradeability

No provisions to upgrade the contracts are found on any of the analyzed contracts.

# Privileged Roles

These are the privileged roles that we identified on each of the audited contracts.

## Ownable

### Owner

The owner of the contract can:

- renounce its ownership.
- transfer the ownership to another address.

Derived contracts may define external or public functions using the `onlyOwner` modifier to make those actions available just to the owner of the contract.

The address doing the deployment is the initial owner of a contract derived from `Ownable`.

## YakBase

This contract is derived from the `Ownable` contract and inherits all its privileged roles. It also adds the dev role and adds capabilities to the owner role defined in `Ownable`.

### Dev

There are no actions defined for the new dev role in the contract code. Derived contracts may define external or public functions using the `onlyDev` modifier to make those actions available just to the owner of the contract.

The address of the initial dev of the contract is passed in the `_settings` contract parameter.

### Owner

Besides all the functionality defined in the `Ownable` contract, the contract owner can enable or disable the deposits.

## YakStrategyV3

This contract is derived from the `YakBase` contract and inherits all its privileged roles. It also adds the EOA role, which corresponds to addresses that do not have deployed code. EOA means "externally owned account".

### EOA

An EOA may trigger the reinvest procedure by invoking the `reinvest()` function. The actual implementation will be in the `_reinvest()` function, which must be implemented by derived contracts.

### Dev

The contract dev may change the dev address to a different one. It must be noted that, unless implemented in derived contracts, the owner of the contract cannot set the address of the developer.

### Owner

Besides all the functionality defined in the `YakBase` contract, the owner can:

- revoke allowances given by the contract to any ERC20 token.
- change the minimum amount of tokens required to trigger the reinvest procedure.

- change the maximum amount of tokens required to force the reinvest procedure on deposit.
- change the dev fee.
- change the reinvest reward.
- rescue funds deployed by the contract. The actual implementation will be in the `_rescueDeployedFunds()` function, which must be implemented by derived contracts. It must be noted that there are two different `rescueDeployedFunds()` functions defined on lines 148 and 235 of `contracts/YakStrategyV3.sol`, both of which invoke the internal `_rescueDeployedFunds()` function.
- recover AVAX and ERC20 tokens owned by the contract.

## VariableRewardsStrategy

It inherits all the roles from the `YakStrategyV3` contract. No new roles were added. Some roles were changed.

### Owner

The contract implements the `_rescueDeployedFunds()` function, allowing the owner to rescue deployed funds. Also all the actions available in the parent contract are available.

### Dev

In the contract constructor, the address of the dev is hardcoded as `0x2D580F9CF2fB2D09BC411532988F2aFdA4E7BefF`. This overrides the setting passed to the constructor in the `_strategySettings` parameter.

Besides all the actions defined for the dev role in the parent contract, it can register and deregister rewards.

# Changelog

- 2022-06-22 – Initial report based on commit `6a57e60e1eebadda288662c627c98a5f2ea9ffd7`.

**Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the YieldYak project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.**