Yearn Finance V3

Yearn Finance V3 represents the latest evolution of the Yearn Finance protocol, a decentralized platform focused on optimizing yield farming strategies. Leveraging advanced automation and smart contract technology, Yearn V3 introduces enhanced modularity, allowing for more flexible and efficient strategy deployments across various DeFi protocols. This iteration aims to provide users with higher returns, reduced risks, and greater customization in managing their assets. By continuously **aggregating yields from different lending protocols**, Yearn V3 ensures that users' assets are **dynamically allocated** to the most profitable opportunities, streamlining the complex process of yield farming.

The Yearn Finance Concepts

- **Yearn Vaults (yVaults)**: A Vault is a Smart Contract that manages users funds and allocates them in different strategies. Funds can be allocated in a **single strategy or in a collection of multiple strategies**. From the documentation page, yVaults are like savings accounts for your crypto assets. They accept your deposit, then route it through strategies which seek out the highest yield available in DeFi. With YearnV3, yVaults are ERC-4626 compatible. See more here.

A vault or "Allocator Vault" in V3 refers to an ERC-4626 compliant contract that takes in user deposits, mints shares corresponding to the user's share of the underlying assets held in that vault, and then allocates the underlying asset to an array of different "strategies" that earn yield on that asset.

- **Stategy**: A strategy in V3 refers to a yield-generating contract added to a vault that has the needed ERC-4626 interface. The strategy takes the underlying asset and deploys it to a single source, generating yield on that asset.

Yearn V3 Main Smart Contracts

Factory Contract

PROF

The factory contract is designed to deploy new vaults using a specific `VAULT_ORIGINAL` as a blueprint. The deployment process ensures that each vault has unique parameters and cannot be duplicated.

Key Function

```
def deploy_new_vault(
    asset: address,
    name: String[64],
    symbol: String[32],
    role_manager: address,
    profit_max_unlock_time: uint256
) -> address
```

This function creates a new vault with the following parameters:

- **asset**: The underlying token the vault will use (e.g., USDC).
- **name**: The name of the vault token (e.g., DeFindex Blend USDC Pool) that will be issued to investors.
- **symbol**: The symbol of the vault token (e.g., dfBlUSDC) that will be issued to investors.
- **role_manager**: The admin responsible for managing the vault's roles.
- **profit_max_unlock_time**: The time over which the profits will unlock (in seconds).

Events

- **NewVault**: Emitted when a new vault is deployed. Provides the vault address
- **UpdateProtocolFeeBps**: Emitted when the protocol fee basis points are updated.
- **UpdateProtocolFeeRecipient**: Emitted when the protocol fee recipient is updated.

NOTE: The vault factory utilizes create2 opcode to deploy vaults to deterministic addresses. This means the same address can not deploy two vaults with the same default parameters for 'asset', 'name' and 'symbol'.

Vault Contract

The vault contract manages user deposits, handles idle assets, and interacts with various strategies to generate yield. It issues vault tokens to users based on their share of the vault's total assets.

Key Concepts

- **vault** (allocator): ERC-4626 compliant contract that accepts deposits, issues shares, and allocates funds to different strategies to earn yield.
- **vault shares**: A tokenized representation of a depositor's share of the underlying balance of a vault.
  strategy: Any ERC-4626 compliant contract that can be added to an allocator vault that earns yield on an underlying asset.
- **debt**: The amount of the underlying asset that an allocator vault has sent to a strategy to earn yield.
- **report**: The function where a vault accounts for any profits or losses a strategy has accrued, charges applicable fees, and locks profit to be distributed to depositors.
- **Idle Amount**: The portion of underlying assets kept liquid within the vault for quick withdrawals.
- **Min Idle Amount**: Minimum liquid amount of underlying assets the vault must maintain.
- **Update Debt**: A function executed by administrators to allocate funds to different strategies, setting target debt levels.
- **Debt Manager**: A role responsible for managing the vault's debt (strategy allocations).
- **Default Queue**: A queue of strategies to take funds from when the vault needs to free up assets. It defines the priority order for liquidating strategy positions.

PROF

Important Functions

- **Deposit**: Users deposit funds into the vault, receiving vault shares in return.
- **Withdraw**: Users burn their vault shares and withdraw their funds, which may involve liquidating strategy positions if the idle amount is insufficient.

Issuing Shares

From the Github Repo:

```
def _total_assets() -> uint256:
    return self.total_idle + self.total_debt

def _deposit(sender: address, recipient: address, assets: uint256) ->
uint256:
    ...
    self.total_idle += assets
    shares: uint256 = self._issue_shares_for_amount(assets, recipient)
    ...

def _issue_shares_for_amount(amount: uint256, recipient: address) ->
uint256:
    total_supply: uint256 = self._total_supply()
    total_assets: uint256 = self._total_assets()
    new_shares: uint256 = 0

    if total_supply == 0:
        new_shares = amount
    elif total_assets > amount:
        new_shares = amount * total_supply / (total_assets - amount)

    if new_shares == 0:
       return 0

    self._issue_shares(new_shares, recipient)
    return new_shares
```

This function calculates and issues new shares based on the amount of assets deposited.

PROF

What these functions do is to mantain the relation between shares and assets invested. In fact, if S_t is the Total Share Supply at time t, A_t is the total amount of Assets at time t, s is the new amount of shares to be minted and a is the amount of assets being invested, what this code is doing is to maintain the following relationship

$$
\frac{S_t}{A_t} = \frac{s}{a}
$$

Because, when `_issue_shares_for_amount` is being called, `total_assets` is alreeady $A_0 + a = A_1$, but `total_supply` is still `S_0` then the relationshit will be

$$
\frac{S_1}{A_1} = \frac{S_0 + s}{A_1} = \frac{s}{a}
$$

$$
(S_0 + s) \cdot a = S_0 \cdot a + s \cdot a = s \cdot A_1
$$

$$
s = \frac{a \cdot S_0 }{A_1 - a}
$$

Links:

- Tech Specs for YearnV3 Vaults
- Vault Management

Strategy Contract

The strategy contract in Yearn V3 focuses on specific yield-generating tasks, delegating standardized ERC-4626 and vault logic to a central `TokenizedStrategy` contract.

Key Components

- **BaseStrategy**: Inherited by strategies to handle communication with the `TokenizedStrategy`.
- **TokenizedStrategy**: Implements all ERC-4626 and vault-specific logic.
- **Modifiers**: Ensure only authorized addresses can call certain functions, enhancing security.

Functions

- **_deployFunds**: Deploys assets into yield sources.
- **_freeFunds**: Frees assets when needed.
- **_harvestAndReport**: Harvests rewards, redeploys idle funds, and reports the strategy's total assets.

Fee and Price Per Share (PPS) Management

Fee Management

PROF

- **Default and Custom Protocol Fees**: The factory contract allows setting default and custom protocol fees for vaults and strategies.
- **Fee Recipient**: Protocol fees are sent to the designated fee recipient, with the remaining fees going to the vault or strategy-specific recipient.

Price Per Share (PPS) Calculation

The PPS is calculated based on the total assets and total supply of shares within the vault.

```
@view
@external
def pricePerShare() -> uint256:
    return self._convert_to_assets(10 ** convert(self.decimals,
```

```
  uint256), Rounding.ROUND_DOWN)
```

This function provides the PPS, ensuring precise asset-to-share conversion.

Calculating Price Per Share (PPS):

- The PPS is a crucial metric for ensuring users receive the correct value for their dfTokens. It is calculated as follows:

$$
\text{PPS} = \frac{\text{Total Assets}}{\text{Total Supply of dfTokens}}
$$

Where:

- **Total Assets**: The sum of the value of assets managed by all adapters plus any idle assets held directly by the DeFindex contract.
- **Total Supply of dfTokens**: The total number of dfTokens issued to users.

To illustrate, consider the following scenario:

- DeFindex has three adapters managing different investments:
  - Adapter A manages $50,000 in a liquidity pool.
  - Adapter B manages $30,000 in a lending pool.
  - Adapter C manages $20,000 in a staking protocol.
- The DeFindex contract holds an additional $10,000 in idle assets.

The Total Assets would be:

$$
50,000 + 30,000 + 20,000 + 10,000 = 110,000 \text{ USDC}
$$

PROF If the Total Supply of dfTokens is 100,000, the PPS would be:

$$
\text{PPS} = \frac{110,000 \text{ USDC}}{100,000 \text{ dfTokens}} = 1.1 \text{ USDC per dfToken}
$$

This calculation ensures users can accurately determine the value of their holdings in DeFindex, promoting transparency and trust.

Problems of Yearn Finance V3

- Does not support multi-asset strategies: For example you can't invest on a vault composed by a strategy of USDC on Aave and another strategy of USDC-WETH on Uniswap.
- Price Per Shares (PPS), ConvertToShares and ConvertToAssets functions need to be described in a single asset.