Armors Labs

Waterfall

Smart Contract Audit

Waterfall Audit Summary

Project name : Waterfall Contract

Project address: None

Code URL : https://github.com/WaterfallDAO/contracts

Commit : dcc85bbb492fade1b77fa49406970ddb5984b8f7

Project target : Waterfall Contract Audit

Blockchain : Binance Smart Chain（BSC）

Test result : PASSED

Audit Info

Audit NO : 0X202204110006

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

Waterfall Audit

The Waterfall team asked us to review and audit their Waterfall contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
Waterfall Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2022-04-11

Audit results

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Waterfall contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc'...)

Armors Labs Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Armors does not guarantee the safety or functionality of the technology agreed to be analyzed.

Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered, deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused. Armors Labs Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

Audited target file

file	md5
./tools/Timelock.sol	61ab7cd048b2fde6e1eb19a463d54f62
./wtf.sol	0cd06749e6140255f1ace5c369c97758
./strategies/Uniswap/UniStrategyLP.sol	1b94ef067a0028cb4c8998bd8a123026
./strategies/Curve/StrategyCurve3TokenPool.sol	241cfccd81d9cd4147a8a668f8bc5203
./strategies/Curve/StrategyCurve2TokenPool.sol	799b6ac20f99ce2913daaba216bdad35
./strategies/Harvest/HarvestStrategyStablecoin.sol	4eef4be1c14ef61bd10823598a9d6d17
./strategies/Harvest/HarvestStrategyUniLP.sol	84908d6116ca9b8f290f7d975f689269
./vaults/Fof.sol	c0ca0454beeeb7840b1ea67455b92b3f
./vaults/Vault.sol	d16b9c7202fe7c4a2054d7f3f8cd2ad9
./vaults/Fund.sol	6d157fb6b9d131342db0fa03fd2bf157
./vaults/WTFFundERC20.sol	170d8ccb602feb2b8e0e3a8d2e57c202
./Migrations.sol	80523b725cb583feec260c84202885cb
./controllers/Controller.sol	0b59588c703876dcfd7b52384de61b7a
./interfaces/IERC20.sol	5e7f70ac5bc528b0fe319edc913ff272
./interfaces/IVault.sol	c7375fa16b1f975f19b9a7c5850ee375
./interfaces/IFof.sol	0b9c430b99fb820679be592413836b99
./yearn/Converter.sol	e5d4e5594de619b97503a18f408f4fad
./yearn/Strategy.sol	854e39ade6fe41fb30ee6f061a1ed989
./yearn/Mintr.sol	30038b64d15db552a8daa460e3b6a342
./yearn/Vault.sol	7fc4bfcb0ed1dca58a27925daa891aee
./yearn/IController.sol	67874a279e8414927c13386d2fe9a6a7
./yearn/OneSplitAudit.sol	b11bdeb7172bfcb31b5c8c921a4e6888
./uniswap/Uni.sol	45c45a5fd9e3a016aaa9b784e8836fe6
./uniswap/TransferHelper.sol	7a2727dd3704e847428c113849da00dd

file	md5
./uniswap/IUniStakingRewards.sol	8d45a12e9f8167e5e3f0a953f3953ddc
./uniswap/IUniPair.sol	f00115ba6e019dc13bbad1d2623b6422
./curve/Gauge.sol	385d9248e485d4157d8bb80a2cbd3709
./curve/Curve.sol	e7eeaf57ce14ab8656f58fafe4ca22e2
./harvest/IHarvestVault.sol	d9c56855e151c5946f2a0161889c9f3f
./harvest/IHarvestPool.sol	45cabb109feba839be6a94199f22fa85
./dforce/Rewards.sol	b5a0eb5ef00e67f11cae84d16710d8ab
./dforce/Token.sol	1d76b9f5d2830f8728bf4203cf06c192
./maker/Maker.sol	758f8b2af0447aad30fac24e11ace468
./weth/WETH.sol	6021678ab3c8767d46cb49f955da7464

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe

Vulnerability	status
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

import "../../../interfaces/curve/Curve.sol";
import "../../../interfaces/curve/Gauge.sol";
import "../../../interfaces/uniswap/Uni.sol";

import "../../../interfaces/yearn/IController.sol";
import "../../../interfaces/yearn/Mintr.sol";

contract StrategyCurve2TokenPool {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    uint256 public constant N_COINS = 2;
    int128 public immutable WANT_COIN_INDEX;
    address public immutable want;
    address public immutable crvLP;
    address public immutable curveDeposit;
    address public immutable gauge;

    address public immutable mintr;
    address public immutable crv;
    address public immutable uni;
    string private name;

    // renBTC, wBTC
    address[N_COINS] public coins;
    uint256[N_COINS] public ZEROS = [uint256(0), uint256(0)];

    uint256 public performanceFee = 500;
    uint256 public immutable performanceMax = 10000;

    uint256 public withdrawalFee = 0;
    uint256 public immutable withdrawalMax = 10000;
    address public governance;
    address public controller;
    address public timelock;

    constructor
    (
        address _controller,
//        string memory _name,
        int128 _wantCoinIndex,
        address[N_COINS] memory _coins,
        address _curveDeposit,
        address _gauge,
        address _crvLP,
        address _crv,
        address _uni,
        address _mintr,
        address _timelock
    )
    public
    {
        governance = msg.sender;
        controller = _controller;
//        name = _name;
        WANT_COIN_INDEX = _wantCoinIndex;
        want = _coins[uint128(_wantCoinIndex)];
        coins = _coins;
        curveDeposit = _curveDeposit;
        gauge = _gauge;
        crvLP = _crvLP;
        crv = _crv;
        uni = _uni;
        mintr = _mintr;
        timelock = _timelock;
    }

    function getName() external view returns (string memory) {
        return name;
    }

    function setWithdrawalFee(uint256 _withdrawalFee) external {
        require(msg.sender == governance, "!governance");
        require(_withdrawalFee < withdrawalMax, "inappropriate withdraw fee");
        withdrawalFee = _withdrawalFee;
    }

    function setPerformanceFee(uint256 _performanceFee) external {
        require(msg.sender == governance, "!governance");
        require(_performanceFee < performanceMax, "inappropriate performance fee");
        performanceFee = _performanceFee;
    }

    function deposit() public {
        _deposit(uint128(WANT_COIN_INDEX));
    }

    function _deposit(uint256 _coinIndex) internal {
        require(_coinIndex < N_COINS, "index exceeded bound");
        address coinAddr = coins[_coinIndex];
        uint256 wantAmount = IERC20(coinAddr).balanceOf(address(this));
        if (wantAmount > 0) {
            IERC20(coinAddr).safeApprove(curveDeposit, 0);
            IERC20(coinAddr).safeApprove(curveDeposit, wantAmount);
            uint256[N_COINS] memory amounts = ZEROS;
            amounts[_coinIndex] = wantAmount;
            // TODO: add minimun mint amount if required
            ICurveDeposit(curveDeposit).add_liquidity(amounts, 0);
        }
        uint256 crvLPAmount = IERC20(crvLP).balanceOf(address(this));
        if (crvLPAmount > 0) {
            IERC20(crvLP).safeApprove(gauge, 0);
            IERC20(crvLP).safeApprove(gauge, crvLPAmount);
            Gauge(gauge).deposit(crvLPAmount);
        }
    }

    // Withdraw all funds, normally used when migrating strategies
    function withdrawAll() external returns (uint256 balance) {
        require(msg.sender == controller, "!controller");
        uint256 _amount = Gauge(gauge).balanceOf(address(this));
        Gauge(gauge).withdraw(_amount);
        IERC20(crvLP).safeApprove(curveDeposit, 0);
        IERC20(crvLP).safeApprove(curveDeposit, _amount);
        // TODO: add minimun mint amount if required
        ICurveDeposit(curveDeposit).remove_liquidity_one_coin(_amount, WANT_COIN_INDEX, 0);

        balance = IERC20(want).balanceOf(address(this));

        address _vault = IController(controller).vaults(address(want));
        require(_vault != address(0), "!vault");
        // additional protection so we don't burn the funds
        IERC20(want).safeTransfer(_vault, balance);
    }

    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _withdrawSome(_amount.sub(_balance));
            _amount = IERC20(want).balanceOf(address(this));
        }
        uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
        IERC20(want).safeTransfer(IController(controller).rewards(), _fee);
        address _vault = IController(controller).vaults(address(want));
        require(_vault != address(0), "!vault");
        // additional protection so we don't burn the funds

        IERC20(want).safeTransfer(_vault, _amount.sub(_fee));
    }

    function _withdrawSome(uint256 _amount) internal {
        uint256 rate = ICurveDeposit(curveDeposit).calc_withdraw_one_coin(10**18, WANT_COIN_INDEX);
        _amount = _amount.mul(10**18).div(rate);

        if (_amount > balanceOfGauge()) {
            _amount = balanceOfGauge();
        }
        Gauge(gauge).withdraw(_amount);
        IERC20(crvLP).safeApprove(curveDeposit, 0);
        IERC20(crvLP).safeApprove(curveDeposit, _amount);
        // TODO: add minimun mint amount if required
        ICurveDeposit(curveDeposit).remove_liquidity_one_coin(_amount, WANT_COIN_INDEX, 0);
    }

    // Controller only function for creating additional rewards from dust
    function withdraw(IERC20 _asset) external returns (uint256 balance) {
        require(msg.sender == controller, "!controller");
        for (uint i = 0; i < N_COINS; ++i) {
            require(coins[i] != address(_asset), "internal token");
        }
        require(crv != address(_asset), "crv");
        require(crvLP != address(_asset), "crvLP");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }

    function harvest(uint _coinIndex,address[] memory path) public {
        require(_coinIndex < N_COINS, "index exceeded bound");
        Mintr(mintr).mint(gauge);
        address harvestingCoin = coins[_coinIndex];
        uint256 _crv = IERC20(crv).balanceOf(address(this));
        if (_crv > 0) {

            IERC20(crv).safeApprove(uni, 0);
            IERC20(crv).safeApprove(uni, _crv);

            Uni(uni).swapExactTokensForTokens(_crv, uint256(0), path, address(this), block.timestamp.
        }
        uint256 harvestAmount = IERC20(harvestingCoin).balanceOf(address(this));
        if (harvestAmount > 0) {
            uint256 _fee = harvestAmount.mul(performanceFee).div(performanceMax);
            IERC20(harvestingCoin).safeTransfer(IController(controller).rewards(), _fee);
            _deposit(_coinIndex);
        }
    }

    function balanceOfWant() public view returns (uint256) {
        return IERC20(want).balanceOf(address(this));
    }

    function balanceOfGauge() public view returns (uint256) {
        return Gauge(gauge).balanceOf(address(this));
    }

    function balanceOfPool() public view returns (uint256) {
        uint256 gaugeBalance = balanceOfGauge();
        // NOTE: this is for curve ren pool only, since calc_withdraw_one_coin
        // would raise error when input 0 amount
        if (gaugeBalance == 0) {
            return 0;
        }
        // portfolio virtual price (for calculating profit) scaled up by 1e18
        uint256 price = ICurveDeposit(curveDeposit).get_virtual_price();
        uint256 _amount = gaugeBalance.mul(price);
        //renBTC and wBTC decmials is 8
        return _amount.div(1e18);
    }

    function balanceOf() public view returns (uint256) {
        return balanceOfWant().add(balanceOfPool());
    }

    function setGovernance(address _governance) external {
        require(msg.sender == timelock, "!timelock");
        governance = _governance;
    }

    function setController(address _controller) external {
        require(msg.sender == timelock, "!timelock");
        controller = _controller;
    }

    function setTimelock(address _timelock) public {
        require(msg.sender == timelock, "!timelock");
        timelock = _timelock;
    }
}

// SPDX-License-Identifier: MIT
```

```
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

import "../../../interfaces/curve/Curve.sol";
import "../../../interfaces/curve/Gauge.sol";
import "../../../interfaces/uniswap/Uni.sol";

import "../../../interfaces/yearn/IController.sol";
import "../../../interfaces/yearn/Mintr.sol";


contract StrategyCurve3TokenPool {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    uint256 public constant N_COINS = 3;
    int128 public immutable WANT_COIN_INDEX;
    address public immutable want;
    address public immutable crvLP;
    address public immutable curveDeposit;
    address public immutable gauge;

    address public immutable mintr;
    address public immutable crv;
    address public immutable uni;
    // used for crv <> weth <> dai route
//    address public immutable weth;
    string private name;

    // DAI, USDC, USDT, TUSD
    address[N_COINS] public coins;
    uint256[N_COINS] public ZEROS = [uint256(0), uint256(0), uint256(0)];

    uint256 public performanceFee = 500;
    uint256 public immutable performanceMax = 10000;

    uint256 public withdrawalFee = 0;
    uint256 public immutable withdrawalMax = 10000;

    address public governance;
    address public controller;
    address public timelock;

    constructor
    (
        address _controller,
        string memory _name,
        int128 _wantCoinIndex,
        address[N_COINS] memory _coins,
        address _curveDeposit,
        address _gauge,
        address _crvLP,
        address _crv,
        address _uni,
        address _mintr,
//        address _weth,
        address _timelock
    )
    public
    {
        governance = msg.sender;
        controller = _controller;
        name = _name;
        WANT_COIN_INDEX = _wantCoinIndex;
        want = _coins[uint128(_wantCoinIndex)];
        coins = _coins;
        curveDeposit = _curveDeposit;
        gauge = _gauge;
        crvLP = _crvLP;
        crv = _crv;
        uni = _uni;
        mintr = _mintr;
//        weth = _weth;
        timelock = _timelock;
    }

    function getName() external view returns (string memory) {
        return name;
    }

    function setWithdrawalFee(uint256 _withdrawalFee) external {
        require(msg.sender == governance, "!governance");
        require(_withdrawalFee < withdrawalMax, "inappropriate withdraw fee");
        withdrawalFee = _withdrawalFee;
    }

    function setPerformanceFee(uint256 _performanceFee) external {
        require(msg.sender == governance, "!governance");
        require(_performanceFee < performanceMax, "inappropriate performance fee");
        performanceFee = _performanceFee;
    }

    function deposit() public {
        _deposit(uint(uint128(WANT_COIN_INDEX)));
    }

    function _deposit(uint256 _coinIndex) internal {
        require(_coinIndex < N_COINS, "index exceeded bound");
        address coinAddr = coins[_coinIndex];
        uint256 wantAmount = IERC20(coinAddr).balanceOf(address(this));
        if (wantAmount > 0) {
            IERC20(coinAddr).safeApprove(curveDeposit, 0);
            IERC20(coinAddr).safeApprove(curveDeposit, wantAmount);
            uint256[N_COINS] memory amounts = ZEROS;
            amounts[_coinIndex] = wantAmount;
            // TODO: add minimun mint amount if required
            ICurveDeposit(curveDeposit).add_liquidity(amounts, 0);
        }
        uint256 crvLPAmount = IERC20(crvLP).balanceOf(address(this));
        if (crvLPAmount > 0) {
            IERC20(crvLP).safeApprove(gauge, 0);
            IERC20(crvLP).safeApprove(gauge, crvLPAmount);
            Gauge(gauge).deposit(crvLPAmount);
        }
    }

    // Withdraw all funds, normally used when migrating strategies
    function withdrawAll() external returns (uint256 balance) {
        require(msg.sender == controller, "!controller");
        uint256 _amount = Gauge(gauge).balanceOf(address(this));
        Gauge(gauge).withdraw(_amount);
        IERC20(crvLP).safeApprove(curveDeposit, 0);
        IERC20(crvLP).safeApprove(curveDeposit, _amount);
        // TODO: add minimun mint amount if required
        ICurveDeposit(curveDeposit).remove_liquidity_one_coin(_amount, WANT_COIN_INDEX, 0);
        balance = IERC20(want).balanceOf(address(this));

        address _vault = IController(controller).vaults(address(want));
        require(_vault != address(0), "!vault");
        // additional protection so we don't burn the funds
        IERC20(want).safeTransfer(_vault, balance);
    }

    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _withdrawSome(_amount.sub(_balance));
            _amount = IERC20(want).balanceOf(address(this));
        }
        uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
        IERC20(want).safeTransfer(IController(controller).rewards(), _fee);
        address _vault = IController(controller).vaults(address(want));
        require(_vault != address(0), "!vault");
        // additional protection so we don't burn the funds

        IERC20(want).safeTransfer(_vault, _amount.sub(_fee));
    }

    function _withdrawSome(uint256 _amount) internal {
        uint256 rate = ICurveDeposit(curveDeposit).calc_withdraw_one_coin(10 ** 18, WANT_COIN_INDEX);
        _amount = _amount.mul(10 ** 18).div(rate);

        if (_amount > balanceOfGauge()) {
            _amount = balanceOfGauge();
        }

        Gauge(gauge).withdraw(_amount);
        IERC20(crvLP).safeApprove(curveDeposit, 0);
        IERC20(crvLP).safeApprove(curveDeposit, _amount);
        // TODO: add minimun mint amount if required
        ICurveDeposit(curveDeposit).remove_liquidity_one_coin(_amount, WANT_COIN_INDEX, 0);
    }

    // Controller only function for creating additional rewards from dust
    function withdraw(IERC20 _asset) external returns (uint256 balance) {
        require(msg.sender == controller, "!controller");
        for (uint i = 0; i < N_COINS; ++i) {
            require(coins[i] != address(_asset), "internal token");
        }
        require(crv != address(_asset), "crv");
        require(crvLP != address(_asset), "crvLP");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }

    function harvest(uint _coinIndex,address[] memory path) public {
        require(_coinIndex < N_COINS, "index exceeded bound");
        Mintr(mintr).mint(gauge);
        address harvestingCoin = coins[_coinIndex];
        uint256 _crv = IERC20(crv).balanceOf(address(this));
        if (_crv > 0) {

            IERC20(crv).safeApprove(uni, 0);
            IERC20(crv).safeApprove(uni, _crv);

            // TODO: add minimun mint amount if required
            Uni(uni).swapExactTokensForTokens(_crv, uint256(0), path, address(this), block.timestamp.
        }
        uint256 harvestAmount = IERC20(harvestingCoin).balanceOf(address(this));
        if (harvestAmount > 0) {
            uint256 _fee = harvestAmount.mul(performanceFee).div(performanceMax);
            IERC20(harvestingCoin).safeTransfer(IController(controller).rewards(), _fee);
            _deposit(_coinIndex);
        }
    }

    function balanceOfWant() public view returns (uint256) {
        return IERC20(want).balanceOf(address(this));
    }

    function balanceOfGauge() public view returns (uint256) {
        return Gauge(gauge).balanceOf(address(this));
    }

    function balanceOfPool() public view returns (uint256) {
        uint256 gaugeBalance = balanceOfGauge();
        // NOTE: this is for curve 3 pool only, since calc_withdraw_one_coin
        // would raise error when input 0 amount
        if (gaugeBalance == 0) {
            return 0;
        }
        // portfolio virtual price (for calculating profit) scaled up by 1e18
        uint256 price = ICurveDeposit(curveDeposit).get_virtual_price();
        uint256 _amount = gaugeBalance.mul(price).div(1e18);
        return convertToCoinAmount(uint128(WANT_COIN_INDEX), _amount);
    }

    function convertToCoinAmount(uint128 _index, uint256 _amount) public view returns (uint256){
        if (_index == 0) {
            return _amount;
        } else if (_index == 1) {
            //usdc decmials is 6
            return _amount.div(1e12);
        }
        //usdt decmials is 6
        return _amount.div(1e12);

    }


    function balanceOf() public view returns (uint256) {
        return balanceOfWant().add(balanceOfPool());
    }

    function setGovernance(address _governance) external {
        require(msg.sender == timelock, "!timelock");
        governance = _governance;
    }

    function setController(address _controller) external {
        require(msg.sender == timelock, "!timelock");
        controller = _controller;
    }

    function setTimelock(address _timelock) public {
        require(msg.sender == timelock, "!timelock");
        timelock = _timelock;
    }
}


// SPDX-License-Identifier: MIT

pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

import "../../../interfaces/yearn/IController.sol";
import "../../../interfaces/harvest/IHarvestVault.sol";
import "../../../interfaces/harvest/IHarvestPool.sol";
import "../../../interfaces/uniswap/Uni.sol";

contract HarvestStrategyStablecoin {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    address public immutable want;
    address public immutable farm;
    address public immutable farmVault;
    address public immutable farmPool;

    address public immutable uni;
    address[] public path;
    string private name;

    uint256 public performanceFee = 500;
    uint256 public immutable performanceMax = 10000;

    uint256 public withdrawalFee = 0;
    uint256 public immutable withdrawalMax = 10000;

    address public governance;
    address public controller;
    address public timelock;

    constructor
    (
        address _controller,
        string memory _name,
        address _want,
        address _farmVault,
        address _farmPool,
        address _farm,
        address _uni,
        address[] memory _path,
        address _timelock
    )
    public
    {
        governance = msg.sender;
        controller = _controller;
        name = _name;
        want = _want;
        farmVault = _farmVault;
        farmPool = _farmPool;
        farm = _farm;
        uni = _uni;
        path = _path;
        timelock = _timelock;
    }

    function getName() external view returns (string memory) {
        return name;
    }

    function setWithdrawalFee(uint256 _withdrawalFee) external {
        require(msg.sender == governance, "!governance");
        require(_withdrawalFee < withdrawalMax, "inappropriate withdraw fee");
        withdrawalFee = _withdrawalFee;
    }

    function setPerformanceFee(uint256 _performanceFee) external {
        require(msg.sender == governance, "!governance");
        require(_performanceFee < performanceMax, "inappropriate performance fee");
        performanceFee = _performanceFee;
    }

    function setPath(address[] memory _path) external {
        require(msg.sender == governance, "!governance");
        path = _path;
    }

    function deposit() public {
        uint256 wantAmount = IERC20(want).balanceOf(address(this));
        if (wantAmount > 0) {
            IERC20(want).safeApprove(farmVault, 0);
            IERC20(want).safeApprove(farmVault, wantAmount);
            IHarvestVault(farmVault).deposit(wantAmount);
        }
        uint256 lpAmount = IERC20(farmVault).balanceOf(address(this));
        if (lpAmount > 0) {
            IERC20(farmVault).safeApprove(farmPool, 0);
            IERC20(farmVault).safeApprove(farmPool, lpAmount);
            IHarvestPool(farmPool).stake(lpAmount);
        }
    }

    // Withdraw all funds, normally used when migrating strategies
    function withdrawAll() external returns (uint256 balance) {
        require(msg.sender == controller, "!controller");
        IHarvestPool(farmPool).exit();
        uint256 _amount = IERC20(farmVault).balanceOf(address(this));
        IHarvestVault(farmVault).withdraw(_amount);

        balance = IERC20(want).balanceOf(address(this));

        address _vault = IController(controller).vaults(address(want));
        require(_vault != address(0), "!vault"); // additional protection so we don't burn the funds
        IERC20(want).safeTransfer(_vault, balance);
    }

    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _withdrawSome(_amount.sub(_balance));
            _amount = IERC20(want).balanceOf(address(this));
        }
        uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
        IERC20(want).safeTransfer(IController(controller).rewards(), _fee);
        address _vault = IController(controller).vaults(address(want));
        require(_vault != address(0), "!vault"); // additional protection so we don't burn the funds

        IERC20(want).safeTransfer(_vault, _amount.sub(_fee));
    }

    function _withdrawSome(uint256 _amount) internal {
        _amount = _amount.mul(10**18).div(IHarvestVault(farmVault).getPricePerFullShare());
        if(_amount > balanceOfPool()) {
            _amount = balanceOfPool();
        }
        IHarvestPool(farmPool).withdraw(_amount);
        IHarvestVault(farmVault).withdraw(_amount);
    }
```

```
// Controller only function for creating additional rewards from dust
function withdraw(IERC20 _asset) external returns (uint256 balance) {
    require(msg.sender == controller, "!controller");
    require(farm != address(_asset), "farm");
    require(farmVault != address(_asset), "farmVault");
    balance = _asset.balanceOf(address(this));
    _asset.safeTransfer(controller, balance);
}

function harvest() public {
    IHarvestPool(farmPool).getReward();
    uint256 _farm = IERC20(farm).balanceOf(address(this));
    if (_farm > 0) {

        IERC20(farm).safeApprove(uni, 0);
        IERC20(farm).safeApprove(uni, _farm);
        // TODO: add minimun mint amount if required
        Uni(uni).swapExactTokensForTokens(_farm, uint256(0), path, address(this), block.timestamp
    }
    uint256 harvestAmount = IERC20(want).balanceOf(address(this));
    if (harvestAmount > 0) {
        uint256 _fee = harvestAmount.mul(performanceFee).div(performanceMax);
        IERC20(want).safeTransfer(IController(controller).rewards(), _fee);
        deposit();
    }
}

function balanceOfWant() public view returns (uint256) {
    return IERC20(want).balanceOf(address(this));
}

function balanceOfPool() public view returns (uint256) {
    return IHarvestPool(farmPool).balanceOf(address(this))
        .mul(IHarvestVault(farmVault).getPricePerFullShare())
        .div(10**18);
}

// TODO: decide whether upgrated through deploying new strategy or using a method
// function upgradeToNewFVault(address _fVault, address _fPool) external {
//     require(msg.sender == timelock, "!timelock");
//     IHarvestPool(farmPool).exit();
//     uint256 _amount = IERC20(farmVault).balanceOf(address(this));
//     IHarvestVault(farmVault).withdraw(_amount);
//     farmVault = _fVault;
//     farmPool = _fPool;
//     deposit();
// }

function balanceOf() public view returns (uint256) {
    return balanceOfWant().add(balanceOfPool());
}

function setGovernance(address _governance) external {
    require(msg.sender == timelock, "!timelock");
    governance = _governance;
}

function setController(address _controller) external {
    require(msg.sender == timelock, "!timelock");
    controller = _controller;
}

function setTimelock(address _timelock) public {
    require(msg.sender == timelock, "!timelock");
    timelock = _timelock;
    }
}

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

import "../../../interfaces/yearn/IController.sol";
import "../../../interfaces/harvest/IHarvestVault.sol";
import "../../../interfaces/harvest/IHarvestPool.sol";
import "../../../interfaces/uniswap/Uni.sol";
import "../../../interfaces/uniswap/IUniPair.sol";

contract HarvestStrategyUniLP {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    address public immutable want;
    address public immutable weth;
    address public immutable underlyingToken;
    address public immutable farm;
    address public immutable farmVault;
    address public immutable farmPool;

    address public immutable uniRouter;
    address[] public pathUnderlying;
    address[] public pathWeth;
    string private name;

    uint256 public performanceFee = 500;
    uint256 public immutable performanceMax = 10000;

    uint256 public withdrawalFee = 0;
    uint256 public immutable withdrawalMax = 10000;

    address public governance;
    address public controller;
    address public timelock;

    constructor
    (
        address _controller,
        string memory _name,
        address _want,
        address _farmVault,
        address _farmPool,
        address _farm,
        address _uniRouter,
        address[] memory _pathWeth,
        address[] memory _pathUnderlying,
        address _timelock
    )
    public
    {
        governance = msg.sender;
        controller = _controller;
        name = _name;
        want = _want;
        farmVault = _farmVault;
        farmPool = _farmPool;
        farm = _farm;
        weth = IUniPair(_want).token0();
        underlyingToken = IUniPair(_want).token1();
        uniRouter = _uniRouter;
        pathWeth = _pathWeth;
        pathUnderlying = _pathUnderlying;
        timelock = _timelock;
    }

    function getName() external view returns (string memory) {
        return name;
    }

    function setWithdrawalFee(uint256 _withdrawalFee) external {
        require(msg.sender == governance, "!governance");
        require(_withdrawalFee < withdrawalMax, "inappropriate withdraw fee");
        withdrawalFee = _withdrawalFee;
    }

    function setPerformanceFee(uint256 _performanceFee) external {
        require(msg.sender == governance, "!governance");
        require(_performanceFee < performanceMax, "inappropriate performance fee");
        performanceFee = _performanceFee;
    }

    function SetPathWeth(address[] memory _path) external {
        require(msg.sender == governance, "!governance");
        pathWeth = _path;
    }

    function SetPathUnderlying(address[] memory _path) external {
        require(msg.sender == governance, "!governance");
        pathUnderlying = _path;
    }

    function deposit() public {
        uint256 wantAmount = IERC20(want).balanceOf(address(this));
        if (wantAmount > 0) {
            IERC20(want).safeApprove(farmVault, 0);
            IERC20(want).safeApprove(farmVault, wantAmount);
            IHarvestVault(farmVault).deposit(wantAmount);
        }
        uint256 lpAmount = IERC20(farmVault).balanceOf(address(this));
        if (lpAmount > 0) {
            IERC20(farmVault).safeApprove(farmPool, 0);
            IERC20(farmVault).safeApprove(farmPool, lpAmount);
            IHarvestPool(farmPool).stake(lpAmount);
        }
    }

    // Withdraw all funds, normally used when migrating strategies
    function withdrawAll() external returns (uint256 balance) {
        require(msg.sender == controller, "!controller");
        IHarvestPool(farmPool).exit();
        uint256 _amount = IERC20(farmVault).balanceOf(address(this));
        IHarvestVault(farmVault).withdraw(_amount);

        balance = IERC20(want).balanceOf(address(this));

        address _vault = IController(controller).vaults(address(want));
        require(_vault != address(0), "!vault");
        // additional protection so we don't burn the funds
        IERC20(want).safeTransfer(_vault, balance);
    }

    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _withdrawSome(_amount.sub(_balance));
            _amount = IERC20(want).balanceOf(address(this));
        }
        uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
        IERC20(want).safeTransfer(IController(controller).rewards(), _fee);
        address _vault = IController(controller).vaults(address(want));
        require(_vault != address(0), "!vault");
        // additional protection so we don't burn the funds

        IERC20(want).safeTransfer(_vault, _amount.sub(_fee));
    }

    function _withdrawSome(uint256 _amount) internal {
        _amount = _amount.mul(10 ** 18).div(IHarvestVault(farmVault).getPricePerFullShare());
        if (_amount > balanceOfPool()) {
            _amount = balanceOfPool();
        }
        IHarvestPool(farmPool).withdraw(_amount);
        IHarvestVault(farmVault).withdraw(_amount);
    }

    // Controller only function for creating additional rewards from dust
    function withdraw(IERC20 _asset) external returns (uint256 balance) {
        require(msg.sender == controller, "!controller");
        require(want != address(_asset), "want");
        require(farm != address(_asset), "farm");
        require(farmVault != address(_asset), "farmVault");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }

    function harvest() public {
        IHarvestPool(farmPool).getReward();
        uint256 _farm = IERC20(farm).balanceOf(address(this));
        if (_farm > 0) {
            IERC20(farm).safeApprove(uniRouter, 0);
            IERC20(farm).safeApprove(uniRouter, _farm);

            Uni(uniRouter).swapExactTokensForTokens(_farm, 1, pathWeth, address(this), block.timestam
            uint256 wethAmount = IERC20(weth).balanceOf(address(this));

            IERC20(weth).safeApprove(uniRouter, 0);
            IERC20(weth).safeApprove(uniRouter, wethAmount);

            wethAmount = wethAmount.div(2);
            Uni(uniRouter).swapExactTokensForTokens(wethAmount, 1, pathUnderlying, address(this), blo
            uint256 underlyingTokenAmount = IERC20(underlyingToken).balanceOf(address(this));

            IERC20(underlyingToken).safeApprove(uniRouter, 0);
            IERC20(underlyingToken).safeApprove(uniRouter, underlyingTokenAmount);

            Uni(uniRouter).addLiquidity(
                weth,
                underlyingToken,
                wethAmount,
                underlyingTokenAmount,
                1, // we are willing to take whatever the pair gives us
                1,
                address(this),
                block.timestamp + 1000
            );
        }
        uint256 harvestAmount = IERC20(want).balanceOf(address(this));
        if (harvestAmount > 0) {
            uint256 _fee = harvestAmount.mul(performanceFee).div(performanceMax);
            IERC20(want).safeTransfer(IController(controller).rewards(), _fee);
            deposit();
        }
    }

    function balanceOfWant() public view returns (uint256) {
        return IERC20(want).balanceOf(address(this));
    }

    function balanceOfPool() public view returns (uint256) {
        return IHarvestPool(farmPool).balanceOf(address(this))
        .mul(IHarvestVault(farmVault).getPricePerFullShare())
        .div(10 ** 18);
    }

    function balanceOf() public view returns (uint256) {
        return balanceOfWant().add(balanceOfPool());
    }

    function setGovernance(address _governance) external {
        require(msg.sender == timelock, "!timelock");
        governance = _governance;
    }

    function setController(address _controller) external {
        require(msg.sender == timelock, "!timelock");
        controller = _controller;
    }

    function setTimelock(address _timelock) public {
        require(msg.sender == timelock, "!timelock");
        timelock = _timelock;
    }
}


// SPDX-License-Identifier: MIT

pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

import "../../../interfaces/yearn/IController.sol";
import "../../../interfaces/uniswap/Uni.sol";
import "../../../interfaces/uniswap/IUniStakingRewards.sol";

contract UniStrategyLP {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    address public immutable want;
    address public immutable weth;
    address public immutable underlyingToken;
    address public immutable rewardUni;
    address public immutable uniRouter;
    address public immutable uniStakingPool;

    address[] public pathUnderlying;
    address[] public pathWeth;
    string private name;
    uint256 public performanceFee = 500;
    uint256 public immutable performanceMax = 10000;

    uint256 public withdrawalFee = 0;
    uint256 public immutable withdrawalMax = 10000;

    address public governance;
    address public controller;
    address public timelock;

    constructor
    (
        address _controller,
        string memory _name,
        address _want,
        address _weth,
        address _underlyingToken,
        address _rewardUni,
        address _uniRouter,
        address _uniStakingPool,
        address[] memory _pathWeth,
        address[] memory _pathUnderlying,
        address _timelock
    )
    public
    {
        governance = msg.sender;
        controller = _controller;
        name = _name;
        want = _want;
        weth = _weth;
        underlyingToken = _underlyingToken;
        rewardUni = _rewardUni;
        uniRouter = _uniRouter;
        uniStakingPool = _uniStakingPool;
        pathWeth = _pathWeth;
        pathUnderlying = _pathUnderlying;
        timelock = _timelock;
    }

    function getName() external view returns (string memory) {
        return name;
    }

    function setWithdrawalFee(uint256 _withdrawalFee) external {
        require(msg.sender == governance, "!governance");
        require(_withdrawalFee < withdrawalMax, "inappropriate withdraw fee");
        withdrawalFee = _withdrawalFee;
    }

    function setPerformanceFee(uint256 _performanceFee) external {
        require(msg.sender == governance, "!governance");
        require(_performanceFee < performanceMax, "inappropriate performance fee");
        performanceFee = _performanceFee;
    }

    function SetPathWeth(address[] memory _path) external {
        require(msg.sender == governance, "!governance");
        pathWeth = _path;
    }

    function SetPathUnderlying(address[] memory _path) external {
        require(msg.sender == governance, "!governance");
        pathUnderlying = _path;
    }
```

```
function deposit() public {
    uint256 wantAmount = IERC20(want).balanceOf(address(this));
    if (wantAmount > 0) {
        IERC20(want).safeApprove(uniStakingPool, 0);
        IERC20(want).safeApprove(uniStakingPool, wantAmount);
        IUniStakingRewards(uniStakingPool).stake(wantAmount);
    }
}

// Withdraw all funds, normally used when migrating strategies
function withdrawAll() external returns (uint256 balance) {
    require(msg.sender == controller, "!controller");
    IUniStakingRewards(uniStakingPool).exit();

    balance = IERC20(want).balanceOf(address(this));
    uint256 rewarAmount = IERC20(rewardUni).balanceOf(address(this));

    address _vault = IController(controller).vaults(address(want));
    require(_vault != address(0), "!vault");
    // additional protection so we don't burn the funds
    IERC20(want).safeTransfer(_vault, balance);
    if (rewarAmount > 0) {
        IERC20(rewardUni).safeTransfer(_vault, rewarAmount);
    }
}

function withdraw(uint256 _amount) external {
    require(msg.sender == controller, "!controller");
    uint256 _balance = IERC20(want).balanceOf(address(this));
    if (_balance < _amount) {
        IUniStakingRewards(uniStakingPool).withdraw(_amount.sub(_balance));
        _amount = IERC20(want).balanceOf(address(this));
    }
    uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
    IERC20(want).safeTransfer(IController(controller).rewards(), _fee);
    address _vault = IController(controller).vaults(address(want));
    require(_vault != address(0), "!vault");
    // additional protection so we don't burn the funds

    IERC20(want).safeTransfer(_vault, _amount.sub(_fee));
}

// Controller only function for creating additional rewards from dust
function withdraw(IERC20 _asset) external returns (uint256 balance) {
    require(msg.sender == controller, "!controller");
    require(want != address(_asset), "want");
    require(rewardUni != address(_asset), "rewardUni");
    require(underlyingToken != address(_asset), "underlyingToken");
    require(weth != address(_asset), "weth");
    balance = _asset.balanceOf(address(this));
    _asset.safeTransfer(controller, balance);
}

function harvest() public {
    IUniStakingRewards(uniStakingPool).getReward();
    uint256 _rewardUni = IERC20(rewardUni).balanceOf(address(this));
    if (_rewardUni > 0) {
        IERC20(rewardUni).safeApprove(uniRouter, 0);
        IERC20(rewardUni).safeApprove(uniRouter, _rewardUni);

        Uni(uniRouter).swapExactTokensForTokens(_rewardUni, 1, pathWeth, address(this), block.tim
        uint256 wethAmount = IERC20(weth).balanceOf(address(this));

        IERC20(weth).safeApprove(uniRouter, 0);
        IERC20(weth).safeApprove(uniRouter, wethAmount);
```

```
            wethAmount = wethAmount.div(2);
            Uni(uniRouter).swapExactTokensForTokens(wethAmount, 1, pathUnderlying, address(this), blo
            uint256 underlyingTokenAmount = IERC20(underlyingToken).balanceOf(address(this));

            IERC20(underlyingToken).safeApprove(uniRouter, 0);
            IERC20(underlyingToken).safeApprove(uniRouter, underlyingTokenAmount);

            Uni(uniRouter).addLiquidity(
                weth,
                underlyingToken,
                wethAmount,
                underlyingTokenAmount,
                1, // we are willing to take whatever the pair gives us
                1,
                address(this), block.timestamp + 1000
            );
        }
        uint256 harvestAmount = IERC20(want).balanceOf(address(this));
        if (harvestAmount > 0) {
            uint256 _fee = harvestAmount.mul(performanceFee).div(performanceMax);
            IERC20(want).safeTransfer(IController(controller).rewards(), _fee);
            deposit();
        }
    }

    function balanceOfWant() public view returns (uint256) {
        return IERC20(want).balanceOf(address(this));
    }

    function balanceOfPool() public view returns (uint256) {
        return IUniStakingRewards(uniStakingPool).balanceOf(address(this));
    }

    function balanceOf() public view returns (uint256) {
        return balanceOfWant().add(balanceOfPool());
    }

    function setGovernance(address _governance) external {
        require(msg.sender == timelock, "!timelock");
        governance = _governance;
    }

    function setController(address _controller) external {
        require(msg.sender == timelock, "!timelock");
        controller = _controller;
    }

    function setTimelock(address _timelock) public {
        require(msg.sender == timelock, "!timelock");
        timelock = _timelock;
    }
}

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

import "../../interfaces/yearn/Converter.sol";
import "../../interfaces/yearn/OneSplitAudit.sol";
import "../../interfaces/yearn/Strategy.sol";
import "../../interfaces/yearn/Vault.sol";

contract Controller {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    address public governance;

    address public onesplit;
    address public rewards;
    address public timelock;

    mapping(address => address) public vaults;
    mapping(address => address) public strategies;
    mapping(address => mapping(address => address)) public converters;

    mapping(address => mapping(address => bool)) public approvedStrategies;

    uint256 public split = 500;
    uint256 public constant max = 10000;

    constructor(address _rewards, address _timelock, address split) public {
        governance = msg.sender;
        onesplit = split;
        rewards = _rewards;
        timelock = _timelock;
    }

    function setRewards(address _rewards) public {
        require(msg.sender == governance, "!governance");
        rewards = _rewards;
    }

    function setSplit(uint256 _split) public {
        require(msg.sender == governance, "!governance");
        require(_split < max, "inappropriate split fee");
        split = _split;
    }

    function setOneSplit(address _onesplit) public {
        require(msg.sender == governance, "!governance");
        onesplit = _onesplit;
    }

    function setGovernance(address _governance) public {
        require(msg.sender == timelock, "!timelock");
        governance = _governance;
    }

    function setTimelock(address _timelock) public {
        require(msg.sender == timelock, "!timelock");
        timelock = _timelock;
    }

    function approveStrategy(address _token, address _strategy) public {
        require(msg.sender == timelock, "!timelock");
        approvedStrategies[_token][_strategy] = true;
    }

    function revokeStrategy(address _token, address _strategy) public {
        require(msg.sender == governance, "!governance");
        approvedStrategies[_token][_strategy] = false;
    }

    function setConverter(
        address _input,
        address _output,
        address _converter
    ) public {
        require(msg.sender == governance, "!governance");
        converters[_input][_output] = _converter;
    }

    function setVault(address _token, address _vault) public {
        require(msg.sender == governance, "!governance");
        require(vaults[_token] == address(0), "vault is 0");
        require(IVault(_vault).token() == _token, "illegal vault");

        vaults[_token] = _vault;
    }

    function setStrategy(address _token, address _strategy) public {
        require(msg.sender == governance, "!governance");
        require(approvedStrategies[_token][_strategy] == true, "!approved");
        require(Strategy(_strategy).want() == _token, "illegal strategy");

        address _current = strategies[_token];
        if (_current != address(0)) {
            Strategy(_current).withdrawAll();
        }
        strategies[_token] = _strategy;
    }

    function earn(address _token, uint256 _amount) public {
        address _strategy = strategies[_token];
        address _want = Strategy(_strategy).want();
        if (_want != _token) {
            address converter = converters[_token][_want];
            IERC20(_token).safeTransfer(converter, _amount);
            _amount = Converter(converter).convert(_strategy);
            IERC20(_want).safeTransfer(_strategy, _amount);
        } else {
            IERC20(_token).safeTransfer(_strategy, _amount);
        }
        Strategy(_strategy).deposit();
    }

    function balanceOf(address _token) external view returns (uint256) {
        return Strategy(strategies[_token]).balanceOf();
    }

    function withdrawAll(address _token) public {
        require(msg.sender == governance, "!governance");
        Strategy(strategies[_token]).withdrawAll();
    }

    function inCaseTokensGetStuck(address _token, uint256 _amount) public {
        require(msg.sender == governance, "!governance");
        IERC20(_token).safeTransfer(msg.sender, _amount);
    }

    function inCaseStrategyTokenGetStuck(address _strategy, address _token) public {
        require(msg.sender == governance, "!governance");
        Strategy(_strategy).withdraw(_token);
    }

    function getExpectedReturn(
        address _strategy,
        address _token,
        uint256 parts
    ) public view returns (uint256 expected) {
        uint256 _balance = IERC20(_token).balanceOf(_strategy);
        address _want = Strategy(_strategy).want();
        (expected,) = OneSplitAudit(onesplit).getExpectedReturn(_token, _want, _balance, parts, 0);
    }

    // Only allows to withdraw non-core strategy tokens ~ this is over and above normal yield
    function yearn(
        address _strategy,
        address _token,
        uint256 parts
    ) public {
        require(msg.sender == governance, "!governance");
        // This contract should never have value in it, but just incase since this is a public call
        uint256 _before = IERC20(_token).balanceOf(address(this));
        Strategy(_strategy).withdraw(_token);
        uint256 _after = IERC20(_token).balanceOf(address(this));
        if (_after > _before) {
            uint256 _amount = _after.sub(_before);
            address _want = Strategy(_strategy).want();
            uint256[] memory _distribution;
            uint256 _expected;
            _before = IERC20(_want).balanceOf(address(this));
            IERC20(_token).safeApprove(onesplit, 0);
            IERC20(_token).safeApprove(onesplit, _amount);
            (_expected, _distribution) = OneSplitAudit(onesplit).getExpectedReturn(_token, _want, _am
            OneSplitAudit(onesplit).swap(_token, _want, _amount, _expected, _distribution, 0);
            _after = IERC20(_want).balanceOf(address(this));
            if (_after > _before) {
                _amount = _after.sub(_before);
                uint256 _reward = _amount.mul(split).div(max);
                earn(_want, _amount.sub(_reward));
                IERC20(_want).safeTransfer(rewards, _reward);
            }
        }
    }

    function withdraw(address _token, uint256 _amount) public {
        require(msg.sender == vaults[_token], "!vault");
        Strategy(strategies[_token]).withdraw(_amount);
    }
}

// SPDX-License-Identifier: MIT

pragma solidity 0.8.4;

import "@openzeppelin/contracts/utils/math/SafeMath.sol";

contract Timelock {
    using SafeMath for uint;

    event NewAdmin(address indexed newAdmin);
    event NewPendingAdmin(address indexed newPendingAdmin);
    event NewDelay(uint indexed newDelay);
    event CancelTransaction(
        bytes32 indexed txHash, address indexed target, uint value, string signature, bytes data, uin
    );
    event ExecuteTransaction(
        bytes32 indexed txHash, address indexed target, uint value, string signature, bytes data, uin
    );
    event QueueTransaction(
        bytes32 indexed txHash, address indexed target, uint value, string signature, bytes data, uin
    );

    uint public constant GRACE_PERIOD = 14 days;
    uint public constant MINIMUM_DELAY = 0;
    // uint public constant MINIMUM_DELAY = 12 hours;
    uint public constant MAXIMUM_DELAY = 30 days;

    address public admin;
    address public pendingAdmin;
    uint public delay;
    bool public admin_initialized;

    mapping(bytes32 => bool) public queuedTransactions;


    constructor(address admin_, uint delay_) public {
        require(delay_ >= MINIMUM_DELAY, "Timelock::constructor: Delay must exceed minimum delay.");
        require(delay_ <= MAXIMUM_DELAY, "Timelock::constructor: Delay must not exceed maximum delay.

        admin = admin_;
        delay = delay_;
        admin_initialized = false;
    }

    // XXX: function() external payable { }
    receive() external payable {}

    function setDelay(uint delay_) public {
        require(msg.sender == address(this), "Timelock::setDelay: Call must come from Timelock.");
        require(delay_ >= MINIMUM_DELAY, "Timelock::setDelay: Delay must exceed minimum delay.");
        require(delay_ <= MAXIMUM_DELAY, "Timelock::setDelay: Delay must not exceed maximum delay.");
        delay = delay_;

        emit NewDelay(delay);
    }

    function acceptAdmin() public {
        require(msg.sender == pendingAdmin, "Timelock::acceptAdmin: Call must come from pendingAdmin.
        admin = msg.sender;
        pendingAdmin = address(0);

        emit NewAdmin(admin);
    }

    function setPendingAdmin(address pendingAdmin_) public {
        // allows one time setting of admin for deployment purposes
        if (admin_initialized) {
            require(msg.sender == address(this), "Timelock::setPendingAdmin: Call must come from Time
        } else {
            require(msg.sender == admin, "Timelock::setPendingAdmin: First call must come from admin.
            admin_initialized = true;
        }
        pendingAdmin = pendingAdmin_;

        emit NewPendingAdmin(pendingAdmin);
    }

    function queueTransaction(
        address target, uint value, string memory signature, bytes memory data, uint eta
    ) public returns (bytes32){
        require(msg.sender == admin, "Timelock::queueTransaction: Call must come from admin.");
        require(
            eta >= getBlockTimestamp().add(delay),
            "Timelock::queueTransaction: Estimated execution block must satisfy delay."
        );

        bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
        queuedTransactions[txHash] = true;

        emit QueueTransaction(txHash, target, value, signature, data, eta);
        return txHash;
    }

    function cancelTransaction(
        address target, uint value, string memory signature, bytes memory data, uint eta
    ) public {
        require(msg.sender == admin, "Timelock::cancelTransaction: Call must come from admin.");

        bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
        queuedTransactions[txHash] = false;

        emit CancelTransaction(txHash, target, value, signature, data, eta);
    }

    function executeTransaction(
        address target, uint value, string memory signature, bytes memory data, uint eta
    ) public payable returns (bytes memory) {
        require(msg.sender == admin, "Timelock::executeTransaction: Call must come from admin.");

        bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
        require(queuedTransactions[txHash], "Timelock::executeTransaction: Transaction hasn't been qu
        require(getBlockTimestamp() >= eta, "Timelock::executeTransaction: Transaction hasn't surpass
        require(getBlockTimestamp() <= eta.add(GRACE_PERIOD), "Timelock::executeTransaction: Transact

        queuedTransactions[txHash] = false;

        bytes memory callData;

        if (bytes(signature).length == 0) {
            callData = data;
        } else {
            callData = abi.encodePacked(bytes4(keccak256(bytes(signature))), data);
        }

        // solium-disable-next-line security/no-call-value
        (bool success, bytes memory returnData) = target.call{value : value}(callData);
        require(success, "Timelock::executeTransaction: Transaction execution reverted.");

        emit ExecuteTransaction(txHash, target, value, signature, data, eta);

        return returnData;
    }

    function getBlockTimestamp() internal view returns (uint) {
        // solium-disable-next-line security/no-block-members
        return block.timestamp;
    }
}


pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";

import "../interfaces/IVault.sol";


contract Fof {
    using SafeERC20 for IERC20;
    using SafeMath for uint256;

    IERC20 public token;
    address public governance;
    address public vault;
    address public fund;

    mapping(uint256 => uint256) public hold;
    mapping(uint256 => uint256) public vaultLP;

    constructor(address _token) public {
        token = IERC20(_token);
        governance = msg.sender;
    }

    function setGovernance(address _governance) public {
        require(governance == msg.sender, "!governance");
        require(_governance != address(0), "is the zero address");
        governance = _governance;
    }

    function setVault(address _vault) public {
        require(governance == msg.sender, "!governance");
        require(_vault != address(0), "is the zero address");
        vault = _vault;
        token.safeApprove(_vault, 2**256 - 1);
    }

    function setFund(address _fund) public {
        require(governance == msg.sender, "!governance");
        require(_fund != address(0), "is the zero address");
        fund = _fund;
    }

    function earn(uint256 _amountA, uint256 _amountB) external {
        require(msg.sender == fund, "Insufficient permissions");
        hold[0] = hold[0].add(_amountA);
        hold[1] = hold[1].add(_amountB);
        if (token.balanceOf(address(this)) > 0) {
            IVault(vault).deposit(token.balanceOf(address(this)));
        }
        distribution();
    }

    function holdProfit(uint256 _amountA, uint256 _amountB) external {
        require(msg.sender == fund, "Insufficient permissions");
        require(_amountA > 0 || _amountB > 0, "Not update");
        hold[0] = hold[0].add(_amountA);
        hold[1] = hold[1].add(_amountB);
        distribution();
    }

    function holdLoss(uint256 _amountA, uint256 _amountB) external {
        require(msg.sender == fund, "Insufficient permissions");
        require(hold[0] >= _amountA && hold[1] >= _amountB, "Insufficient balance");
        hold[0] = hold[0].sub(_amountA);
        hold[1] = hold[1].sub(_amountB);
        distribution();
    }

    function distribution() private {
        uint256 vlp = ERC20(vault).balanceOf(address(this));
        uint256 total = hold[0].add(hold[1]) == 0 ? 1 : hold[0].add(hold[1]);
        vaultLP[0] = hold[0].mul(vlp).div(total);
        vaultLP[1] = vlp.sub(vaultLP[0]);
    }

    function balance() public view returns (uint256){
        return IVault(vault).balance().add(token.balanceOf(address(this)));
    }

    function withdraw(uint256 _fid, uint256 _amount) external {
        require(msg.sender == fund, "Insufficient permissions");
        require(hold[_fid] >= _amount, "Insufficient balance");
        if (token.balanceOf(address(this)) >= _amount) {
            token.safeTransfer(fund, _amount);
        } else {
            uint256 real = _amount.sub(token.balanceOf(address(this)));
            uint256 share = real.mul(vaultLP[_fid]).div(hold[_fid]);
            IVault(vault).withdraw(share);
            token.safeTransfer(fund, _amount);
        }
    }

    function withdrawAll() external {
        require(msg.sender == governance, "Insufficient permissions");
        IVault(vault).withdrawAll();
    }

}


pragma solidity ^0.8.4;

import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

import "../../interfaces/uniswap/TransferHelper.sol";
import "./WTFFundERC20.sol";
import "../interfaces/IFof.sol";

contract Fund is ReentrancyGuard {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;
    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private stake0Address;
    EnumerableSet.AddressSet private redeem0Address;
    EnumerableSet.AddressSet private stake1Address;
    EnumerableSet.AddressSet private redeem1Address;
    IERC20 public token;
    WTFFundERC20 public tokenA;
    WTFFundERC20 public tokenB;


    uint256 public cycle;
    uint256 public constant DENOMINATOR = 1000;
    uint256 public numerator;

    address public governance;
    address public fof;


    enum poolStatus{
        pending,
        settlement,
        investment
    }

    struct Account {
        uint256 stake;
        uint256 redeem;
        uint256 redeemReserve;
    }

    struct FundPool {
        uint256 currentStake;
        uint256 currentLPToken;
        uint256 mirrorStake;
        uint256 currentRedeem;
        uint256 totalRedeem;
        uint256 currentAmount;
        uint256 lastBillingCycle;
        poolStatus status;
    }

    mapping(address => mapping(uint256 => Account)) public account_;
    mapping(uint256 => FundPool) public fund_;

    event Deposit(address indexed user, uint256 indexed fid, uint256 amount);
    event WithdrawStake(address indexed user, uint256 indexed fid, uint256 amount);
    event WithdrawAmount(address indexed user, uint256 indexed fid, uint256 amount);
    event Claim(address indexed user, uint256 indexed fid, uint256 amount);
    event CancelWithdraw(address indexed user, uint256 indexed fid, uint256 amount);
    event Profit(uint256 amountA, uint256 amountB);
    event Loss(uint256 amountA, uint256 amountB);

    constructor(
        string memory _nameA,
        string memory _symbolA,
        string memory _nameB,
        string memory _symbolB,
        address _token,
        uint256 _cycle,
        address _fof
    ) public {
        tokenA = new WTFFundERC20(_nameA, _symbolA, ERC20(_token).decimals());
        tokenB = new WTFFundERC20(_nameB, _symbolB, ERC20(_token).decimals());
        token = IERC20(_token);
        cycle = _cycle;
        fof = _fof;
        governance = msg.sender;
    }


    function initialize() public {
        require(msg.sender == governance, "!governance");
        FundPool storage fundA = fund_[0];
        FundPool storage fundB = fund_[1];
        fundA.lastBillingCycle = block.timestamp;
        fundB.lastBillingCycle = block.timestamp;
        fundA.status = poolStatus.pending;
        fundB.status = poolStatus.pending;
    }

    function setGovernance(address _governance) public {
        require(msg.sender == governance, "!governance");
        require(_governance != address(0), "Fund: is the zero address");
        governance = _governance;
    }

    function setCycle(uint256 _cycle) public {
        require(msg.sender == governance, "!governance");
        cycle = _cycle;
    }


    function setNumerator(uint256 _newNumerator) public {
        require(msg.sender == governance, "!governance");
        require(_newNumerator <= DENOMINATOR, "Out of range");
        numerator = _newNumerator;
    }


    function setFof(address _fof) public {
        require(msg.sender == governance, "!governance");
        require(_fof != address(0), "Fund: is the zero address");
        fof = _fof;
    }

    function getPoolStatus() public view returns (poolStatus){
        FundPool memory fundA = fund_[0];
        return fundA.status;
    }

    function getActiveLength(EnumerableSet.AddressSet storage _address) private view returns (uint256
        return EnumerableSet.length(_address);
    }

    function getLength() public view returns (uint256, uint256, uint256, uint256){
        require(msg.sender == governance, "!governance");
        uint256 stakeA = EnumerableSet.length(stake0Address);
        uint256 stakeB = EnumerableSet.length(stake1Address);
        uint256 redeemA = EnumerableSet.length(redeem0Address);
        uint256 redeemAB = EnumerableSet.length(redeem1Address);
        return (stakeA, stakeB, redeemA, redeemAB);
    }


    function deposit(uint256 _fid, uint256 _amount) public nonReentrant {
        Account storage user = account_[msg.sender][_fid];
        FundPool storage pool = fund_[_fid];
        require(pool.status == poolStatus.pending, "!pending");
        WTFFundERC20 wtfToken = _fid == 0 ? tokenA : tokenB;
        uint256 _before = token.balanceOf(address(this));
        IERC20(token).safeTransferFrom(msg.sender, address(this), _amount);
        uint256 _after = token.balanceOf(address(this));
        _amount = _after.sub(_before);
        if (wtfToken.totalSupply() == 0) {
            wtfToken.mint(msg.sender, _amount);
            pool.currentLPToken = pool.currentLPToken.add(_amount);
        } else {
            uint256 amount = _amount.mul(wtfToken.totalSupply()).div(pool.currentAmount.add(pool.curr
            wtfToken.mint(msg.sender, amount);
            pool.currentLPToken = pool.currentLPToken.add(amount);
        }

        user.stake = user.stake.add(_amount);

        pool.currentStake = pool.currentStake.add(_amount);
        pool.mirrorStake = pool.mirrorStake.add(_amount);
        EnumerableSet.AddressSet storage stakeAddress = _fid == 0 ? stake0Address : stake1Address;
        EnumerableSet.add(stakeAddress, msg.sender);
        emit Deposit(msg.sender, _fid, _amount);
    }

    function withdraw(uint256 _fid, uint256 _amount) public nonReentrant {
        Account storage user = account_[msg.sender][_fid];
        FundPool storage pool = fund_[_fid];
        require(pool.status == poolStatus.pending, "!pending");
        WTFFundERC20 wtfToken = _fid == 0 ? tokenA : tokenB;
        require(wtfToken.balanceOf(msg.sender) >= _amount, "Insufficient balance");
        TransferHelper.safeTransferFrom(address(wtfToken), msg.sender, address(this), _amount);
        EnumerableSet.AddressSet storage stakeAddress = _fid == 0 ? stake0Address : stake1Address;
        EnumerableSet.AddressSet storage redeemAddress = _fid == 0 ? redeem0Address : redeem1Address;
        if (user.stake > 0) {

            uint256 amount = user.stake.mul(wtfToken.totalSupply()).div(pool.currentAmount.add(pool.c
            if (amount >= _amount) {
                uint256 redeem = _amount.mul(pool.currentAmount.add(pool.currentStake)).div(wtfToken.
                user.stake = user.stake.sub(redeem);
                if (user.stake <= 0) {
                    EnumerableSet.remove(stakeAddress, msg.sender);
                }
                pool.currentStake = pool.currentStake.sub(redeem);
                pool.mirrorStake = pool.mirrorStake.sub(redeem);
                IERC20(token).safeTransfer(msg.sender, redeem);
                emit WithdrawStake(msg.sender, _fid, redeem);
                wtfToken.burn(_amount);
                pool.currentLPToken = pool.currentLPToken.sub(_amount);
            } else {
                pool.currentStake = pool.currentStake.sub(user.stake);
                pool.mirrorStake = pool.mirrorStake.sub(user.stake);
                IERC20(token).safeTransfer(msg.sender, user.stake);
                emit WithdrawStake(msg.sender, _fid, user.stake);
                wtfToken.burn(amount);
                pool.currentLPToken = pool.currentLPToken.sub(amount);
                user.stake = 0;
                EnumerableSet.remove(stakeAddress, msg.sender);
                uint256 result = _amount.sub(amount);
                user.redeem = user.redeem.add(result);
                pool.currentRedeem = pool.currentRedeem.add(result);
                EnumerableSet.add(redeemAddress, msg.sender);
            }
        } else {
            user.redeem = user.redeem.add(_amount);
            pool.currentRedeem = pool.currentRedeem.add(_amount);
            EnumerableSet.add(redeemAddress, msg.sender);
        }
    }


    function isPending() public view returns (bool){
        if (fund_[0].lastBillingCycle.add(cycle) <= block.timestamp) {
            return true;
        }
        return false;
    }

    function startSettle() public {
        require(msg.sender == governance, "!governance");
        require(isPending(), "time is not up yet");
        FundPool storage fundA = fund_[0];
        FundPool storage fundB = fund_[1];
        fundA.status = poolStatus.settlement;
        fundB.status = poolStatus.settlement;
    }

    function startInvest() public {
        require(msg.sender == governance, "!governance");
        require(isPending(), "time is not up yet");
        FundPool storage fundA = fund_[0];
        FundPool storage fundB = fund_[1];
        fundA.status = poolStatus.investment;
        fundB.status = poolStatus.investment;
    }

    function startPending() public {
        require(msg.sender == governance, "!governance");
        FundPool storage fundA = fund_[0];
        FundPool storage fundB = fund_[1];
        require(fundA.status != poolStatus.pending, "Don't repeat");
        require(fundA.currentRedeem == 0 && fundA.currentStake == 0);
        require(fundB.currentRedeem == 0 && fundB.currentStake == 0);
        fundA.status = poolStatus.pending;
        fundB.status = poolStatus.pending;

        fundA.lastBillingCycle = block.timestamp;
        fundB.lastBillingCycle = block.timestamp;
    }

    function earn(uint256 lengthA, uint256 lengthB) public {
        require(msg.sender == governance, "!governance");
        FundPool storage fundA = fund_[0];
        FundPool storage fundB = fund_[1];
        require(fundA.status == poolStatus.investment, "!investment");
        uint256 amountA = fundA.currentStake;
        uint256 amountB = fundB.currentStake;
        uint256 redeem = fundA.totalRedeem.add(fundB.totalRedeem);
        require(amountA > 0 || amountB > 0, "No new investment found");
        uint256 amountAReserve = 0;
        uint256 amountBReserve = 0;
        if (amountA > 0) {
            uint256 length = getActiveLength(stake0Address);
            require(lengthA <= length, "Insufficient users");
            while (lengthA > 0) {
                address sAddress = EnumerableSet.at(stake0Address, 0);
                Account storage user = account_[sAddress][0];
                amountAReserve += user.stake;
                user.stake = 0;
                EnumerableSet.remove(stake0Address, sAddress);
                lengthA--;
            }
        }
        if (amountB > 0) {
            uint256 length = getActiveLength(stake1Address);
            require(lengthB <= length, "Insufficient users");
            while (lengthB > 0) {
                address sAddress = EnumerableSet.at(stake1Address, 0);
                Account storage user = account_[sAddress][1];
                amountBReserve += user.stake;
                user.stake = 0;
                EnumerableSet.remove(stake1Address, sAddress);
                lengthB--;
            }
        }
        uint256 balance = token.balanceOf(address(this));
        uint256 remainder = balance > redeem ? balance.sub(redeem) : 0;
        if (remainder > 0) {
            token.safeTransfer(fof, remainder);
        }
        IFof(fof).earn(amountAReserve, amountBReserve);
        fundA.currentAmount = fundA.currentAmount.add(amountAReserve);
        fundB.currentAmount = fundB.currentAmount.add(amountBReserve);
        fundA.currentStake = fundA.currentStake.sub(amountAReserve);
        fundB.currentStake = fundB.currentStake.sub(amountBReserve);
        if (fundA.currentStake == 0 && fundB.currentStake == 0) {
            fundA.mirrorStake = 0;
            fundB.mirrorStake = 0;
        }
        if (fundA.currentRedeem == 0 && fundB.currentRedeem == 0) {
            fundA.currentLPToken = 0;
            fundB.currentLPToken = 0;
        }
    }
```

```
function handleWithdrawFundA(uint256 lengthA) public {
    require(msg.sender == governance, "!governance");
    FundPool storage fundA = fund_[0];
    require(fundA.status == poolStatus.settlement, "!settlement");
    uint256 amountA = fundA.currentRedeem;
    require(amountA > 0, "No new redeem found");
    uint256 amountAReserve = 0;
    uint256 redeemAReserve = 0;
    if (amountA > 0) {
        uint256 length = getActiveLength(redeem0Address);
        require(lengthA <= length, "Insufficient users");
        while (lengthA > 0) {
            address rAddress = EnumerableSet.at(redeem0Address, 0);
            Account storage user = account_[rAddress][0];

            require(tokenA.totalSupply() > fundA.currentLPToken, "check earn");
            uint256 reserve = user.redeem.mul(fundA.currentAmount).div(tokenA.totalSupply().sub(f
            amountAReserve += reserve;
            redeemAReserve += user.redeem;
            user.redeemReserve = user.redeemReserve.add(reserve);
            fundA.currentRedeem = fundA.currentRedeem.sub(user.redeem);
            user.redeem = 0;
            EnumerableSet.remove(redeem0Address, rAddress);
            lengthA--;
        }
        fundA.currentAmount = fundA.currentAmount.sub(amountAReserve);
        tokenA.burn(redeemAReserve);
    }
    if (fundA.mirrorStake >= amountAReserve) {
        fundA.totalRedeem = fundA.totalRedeem.add(amountAReserve);
        fundA.mirrorStake = fundA.mirrorStake.sub(amountAReserve);
        IFof(fof).holdLoss(amountAReserve, 0);
    } else {
        IFof(fof).withdraw(0, amountAReserve.sub(fundA.mirrorStake));
        fundA.mirrorStake = 0;
        fundA.totalRedeem = fundA.totalRedeem.add(amountAReserve);
        IFof(fof).holdLoss(amountAReserve, 0);
    }
    if (fundA.currentRedeem <= 0) {
        fundA.currentLPToken = 0;
    }
}


function handleWithdrawFundB(uint256 lengthB) public {
    require(msg.sender == governance, "!governance");
    FundPool storage fundB = fund_[1];
    require(fundB.status == poolStatus.settlement, "!settlement");
    uint256 amountB = fundB.currentRedeem;
    require(amountB > 0, "No new redeem found");
    uint256 amountBReserve = 0;
    uint256 redeemBReserve = 0;
    if (amountB > 0) {
        uint256 length = getActiveLength(redeem1Address);
        require(lengthB <= length, "Insufficient users");
        while (lengthB > 0) {
            address rAddress = EnumerableSet.at(redeem1Address, 0);
            Account storage user = account_[rAddress][1];
            require(tokenB.totalSupply() > fundB.currentLPToken, "check earn");
            uint256 reserve = user.redeem.mul(fundB.currentAmount).div(tokenB.totalSupply().sub(f
            amountBReserve += reserve;
            redeemBReserve += user.redeem;
            user.redeemReserve = user.redeemReserve.add(reserve);
            fundB.currentRedeem = fundB.currentRedeem.sub(user.redeem);
            user.redeem = 0;
```

```
                EnumerableSet.remove(redeem1Address, rAddress);
                lengthB--;
            }
            fundB.currentAmount = fundB.currentAmount.sub(amountBReserve);
            tokenB.burn(redeemBReserve);
        }
        if (fundB.currentStake >= amountBReserve) {
            fundB.totalRedeem = fundB.totalRedeem.add(amountBReserve);
            fundB.mirrorStake = fundB.mirrorStake.sub(amountBReserve);
            IFof(fof).holdLoss(0, amountBReserve);
        } else {
            IFof(fof).withdraw(1, amountBReserve.sub(fundB.mirrorStake));
            fundB.mirrorStake = 0;
            fundB.totalRedeem = fundB.totalRedeem.add(amountBReserve);
            IFof(fof).holdLoss(0, amountBReserve);
        }
        if (fundB.currentRedeem <= 0) {
            fundB.currentLPToken = 0;
        }
    }


    function claim(uint256 _fid) public {
        Account storage user = account_[msg.sender][_fid];
        FundPool storage pool = fund_[_fid];
        require(pool.status != poolStatus.settlement, "is settlement");
        require(user.redeemReserve > 0, "No amount to collect");
        token.safeTransfer(msg.sender, user.redeemReserve);
        emit Claim(msg.sender, _fid, user.redeemReserve);
        pool.totalRedeem = pool.totalRedeem.sub(user.redeemReserve);
        emit WithdrawAmount(msg.sender, _fid, user.redeemReserve);
        user.redeemReserve = 0;
    }

    function cancelWithdraw(uint256 _fid, uint256 _amount) public {
        Account storage user = account_[msg.sender][_fid];
        FundPool storage pool = fund_[_fid];
        require(pool.status == poolStatus.pending, "!pending");
        WTFFundERC20 token = _fid == 0 ? tokenA : tokenB;
        EnumerableSet.AddressSet storage stakeAddress = _fid == 0 ? stake0Address : stake1Address;
        require(user.redeem >= _amount && pool.currentRedeem >= _amount, "Abnormal amount");
        user.redeem = user.redeem.sub(_amount);
        if (user.redeem <= 0) {
            EnumerableSet.remove(stakeAddress, msg.sender);
        }
        pool.currentRedeem = pool.currentRedeem.sub(_amount);
        TransferHelper.safeTransfer(address(token), msg.sender, _amount);
        emit CancelWithdraw(msg.sender, _fid, _amount);
    }

    function settle() public {
        require(msg.sender == governance, "!governance");
        FundPool storage fundA = fund_[0];
        FundPool storage fundB = fund_[1];
        require(fundB.status == poolStatus.settlement, "!Settlement");
        uint256 fofAssets = IFof(fof).balance();
        uint256 total = fundA.currentAmount.add(fundB.currentAmount);
        require(fofAssets != total, "no change");
        if (fofAssets > total) {
            (uint256 A,uint256 B) = profit(fofAssets.sub(total));
            IFof(fof).holdProfit(A, B);
            emit Profit(A, B);
        } else {
            (uint256 A,uint256 B) = loss(total.sub(fofAssets));
            IFof(fof).holdLoss(A, B);
            emit Loss(A, B);
        }
    }

    function profit(uint256 _amount) private returns (uint256, uint256) {
        FundPool storage fundA = fund_[0];
        FundPool storage fundB = fund_[1];
        uint256 profitA = fundA.currentAmount.mul(_amount).div(fundA.currentAmount.add(fundB.currentA
        uint256 rewardB = profitA.mul(numerator).div(DENOMINATOR);

        uint256 realA = profitA.sub(rewardB);

        fundA.currentAmount = fundA.currentAmount.add(realA);

        uint256 realB = _amount.sub(profitA).add(rewardB);
        fundB.currentAmount = fundB.currentAmount.add(realB);
        return (realA, realB);
    }


    function loss(uint256 _amount) private returns (uint256, uint256) {

        FundPool storage fundA = fund_[0];
        FundPool storage fundB = fund_[1];
        uint256 lossA = fundA.currentAmount.mul(_amount).div(fundA.currentAmount.add(fundB.currentAmo
        if (lossA >= fundB.currentAmount) {
            uint256 realA = lossA.sub(fundB.currentAmount);
            uint256 lossB = fundB.currentAmount;
            fundA.currentAmount = fundA.currentAmount.sub(realA);
            fundB.currentAmount = 0;
            return (lossA, lossB);
        } else {
            uint256 lossB = _amount;
            if (_amount >= fundB.currentAmount) {
                fundB.currentAmount = 0;
            } else {
                fundB.currentAmount = fundB.currentAmount.sub(_amount);
            }
            return (0, lossB);
        }
    }
}

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

import "../../interfaces/yearn/IController.sol";

contract Vault is ERC20, ReentrancyGuard {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    IERC20 public token;

    uint256 public min = 9500;
    uint256 public constant max = 10000;
    address public governance;
    address public controller;
    address public timelock;

    constructor(address _token, address _controller, string memory _name, string memory _symbol, addr
        public
        ERC20(_name, _symbol)
    {
//          _setupDecimals(ERC20(_token).decimals());
        token = IERC20(_token);
        governance = msg.sender;
        controller = _controller;
        timelock = _timelock;
    }

    function balance() public view returns (uint256) {
        return token.balanceOf(address(this)).add(IController(controller).balanceOf(address(token)));
    }

    function setMin(uint256 _min) external {
        require(msg.sender == governance, "!governance");
        require(_min < max, "inappropriate min reserve token amount");
        min = _min;
    }

    function setGovernance(address _governance) public {
        require(msg.sender == timelock, "!timelock");
        governance = _governance;
    }

    function setController(address _controller) public {
        require(msg.sender == timelock, "!timelock");
        controller = _controller;
    }

    function setTimelock(address _timelock) public {
        require(msg.sender == timelock, "!timelock");
        timelock = _timelock;
    }

    // Custom logic in here for how much the vault allows to be borrowed
    // Sets minimum required on-hand to keep small withdrawals cheap
    function available() public view returns (uint256) {
        return token.balanceOf(address(this)).mul(min).div(max);
    }

    function earn() public {
        require(msg.sender == governance, "!governance");
        uint256 _bal = available();
        token.safeTransfer(controller, _bal);
        IController(controller).earn(address(token), _bal);
    }

    function depositAll() external {
        deposit(token.balanceOf(msg.sender));
    }

    function deposit(uint256 _amount) public nonReentrant {
        uint256 _pool = balance();
        uint256 _before = token.balanceOf(address(this));
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 _after = token.balanceOf(address(this));
        _amount = _after.sub(_before); // Additional check for deflationary tokens
        uint256 shares = 0;
        if (totalSupply() == 0) {
            shares = _amount;
        } else {
            shares = (_amount.mul(totalSupply())).div(_pool);
        }
        _mint(msg.sender, shares);
    }

    function withdrawAll() external {
        withdraw(balanceOf(msg.sender));
    }

    // Used to swap any borrowed reserve over the debt limit to liquidate to 'token'
    function harvest(address reserve, uint256 amount) external {
        require(msg.sender == controller, "!controller");
        require(reserve != address(token), "token");
        IERC20(reserve).safeTransfer(controller, amount);
    }

    // No rebalance implementation for lower fees and faster swaps
    function withdraw(uint256 _shares) public nonReentrant{
        uint256 r = (balance().mul(_shares)).div(totalSupply());
        _burn(msg.sender, _shares);

        // Check balance
        uint256 b = token.balanceOf(address(this));
        if (b < r) {
            uint256 _withdraw = r.sub(b);
            IController(controller).withdraw(address(token), _withdraw);
            uint256 _after = token.balanceOf(address(this));
            uint256 _diff = _after.sub(b);
            if (_diff < _withdraw) {
                r = b.add(_diff);
            }
        }

        token.safeTransfer(msg.sender, r);
    }

    function getPricePerFullShare() public view returns (uint256) {
        if (totalSupply() > 0) {
            return balance().mul(1e18).div(totalSupply());
        }
        return 0;
    }
}


pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";


contract WTFFundERC20 is ERC20, Ownable {

    constructor(string memory name, string memory symbol, uint8 decimals_) public ERC20(name, symbol)
        //        _setupDecimals(decimals_);
    }
    function mint(address _to, uint256 _amount) onlyOwner external {
        _mint(_to, _amount);
    }

    function burn(uint256 _amount) external {
        _burn(msg.sender, _amount);
    }
}
```

```
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract wtf is ERC20 {
    uint256 public constant MAX_SUPPLY = 13 * 1e8 * 1e18;
    constructor()  ERC20("Waterfall", "WTF") {
        super._mint(msg.sender, MAX_SUPPLY);
    }
}
```

Analysis of audit results

Re-Entrancy

- **Description:**
One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Arithmetic Over/Under Flows

- **Description:**
The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Unexpected Blockchain Currency

- **Description:**
Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a

contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Delegatecall

- **Description:**
The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devestating vulnerabilities in smart contracts as will be discussed in this section.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Entropy Illusion

- **Description:**
All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send())

fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Race Conditions / Front Running

- **Description:**
The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Denial Of Service (DOS)

- **Description:**
This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Block Timestamp Manipulation

- **Description:**
Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Constructors with Care

- **Description:**
Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Unintialised Storage Pointers

- **Description:**
The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

Floating Points and Numerical Precision

- **Description:**
As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
no.

tx.origin Authentication

- **Description:**
Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

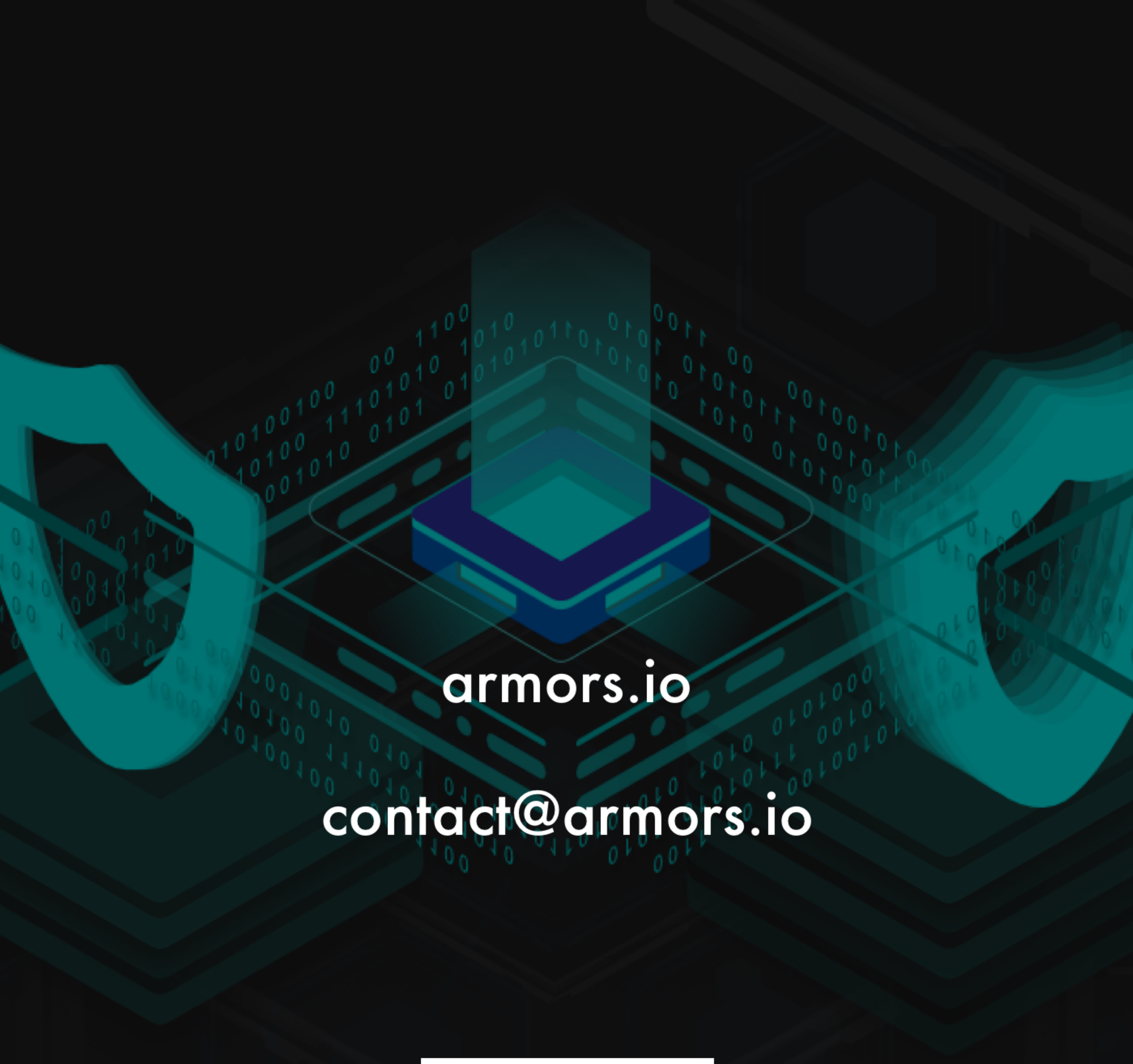
- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.



armors.io

contact@armors.io