Smart Contract Security Audit Report

The SlowMist Security Team received the Cook Index team's application for smart contract security audit of the SingleIndexModule, TradeModule, UniswapPairPriceAdapter, WrapModule. on March 08, 2021. The following are the details and results of this smart contract security audit:

Project name :

Cook Index

Project Address:

UniswapPairPriceAdapter.sol

Github:

https://github.com/CookFinance/cook-index/blob/main/cook-protocol-contracts/contracts/protocol/integration/UniswapPairPriceAdapter.sol

commit: bdfa6fb4e4261887d6d6c09bc2cd1085e98e9d74

SingleIndexModule.sol

Github:

https://github.com/CookFinance/cook-index/blob/main/cook-protocol-contracts/contracts/protocol/modules/SingleIndexModule.sol

commit: c7c89b78d97e672a2bd8e046de5d0f7bb3643ae8

TradeModule.sol

Github:

https://github.com/CookFinance/cook-index/blob/main/cook-protocol-contracts/contracts/protocol/modules/TradeModule.sol

commit: c7c89b78d97e672a2bd8e046de5d0f7bb3643ae8

WrapModule.sol

Github:

https://github.com/CookFinance/cook-index/blob/main/cook-protocol-contracts/contracts/protocol/modules/WrapModule.sol

commit: c7c89b78d97e672a2bd8e046de5d0f7bb3643ae8

The audit items and results :

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

No.	Audit Items	Audit Subclass	Audit Subclass Result
1	Overflow Audit	-	Passed
2	Race Conditions Audit	-	Passed
3	Authority Control Audit	Permission vulnerability audit	Passed
		Excessive authority audit	Passed
4	Safety Design Audit	Zeppelin module safe use	Passed
		Compiler version security	Passed
		Hard-coded address security	Passed
		Fallback function safe use	Passed
		Show coding security	Passed
		Function return value security	Passed
		Call function security	Passed
5	Denial of Service Audit	-	Passed
6	Gas Optimization Audit	-	Passed
7	Design Logic Audit	-	Passed
8	“False Deposit” vulnerability Audit	-	Passed
9	Malicious Event Log Audit	-	Passed
10	Scoping and Declarations Audit	-	Passed
11	Replay Attack Audit	ECDSA's Signature Replay Audit	Passed
12	Uninitialized Storage Pointers Audit	-	Passed
13	Arithmetic Accuracy Deviation Audit	-	Passed

Audit Result : **Passed**

Audit Number : 0X002103180004

Audit Date : Mar. 18, 2021

Audit Team : SlowMist Security Team

Summary: We audited the SingleIndexModule, Trademodule, WarpModule and UniswapPairPriceAdapter module of Cook Index project. During the audit, we found the following issues:

1. **In SingleIndexModule, the trade function does not set a slippage protection when trade, which may results in sandwich attack.**
2. **The initialize function of SingleIndexModule, TradeModule and WarpModule were not restricted to can only be called once.**
3. **The trade function of TradeModule does not check if _receiveToken is allowed to trade.**

After communicating with the Cook Index team, the above issue are ignored because the following reasons:

1. **For issue one, there is a trade amount limitation that guarantee the trade amount is small enough to ignore the slippage issue.**
2. **For issue two, the initialize functoion can not be called again without removing the module first.**
3. **For issue three, this is a feature, the address is up the stack with a manager contract.**

The source code:

SingleIndexModule.sol

```
/*
    Copyright 2021 Cook Finance.

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.

    SPDX-License-Identifier: Apache License, Version 2.0
*/

pragma solidity 0.6.10;
pragma experimental "ABIEncoderV2";

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { Math } from "@openzeppelin/contracts/math/Math.sol";
import { ReentrancyGuard } from "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import { SafeCast } from "@openzeppelin/contracts/utils/SafeCast.sol";
import { SafeMath } from "@openzeppelin/contracts/math/SafeMath.sol";

import { AddressArrayUtils } from "../../lib/AddressArrayUtils.sol";
import { IController } from "../../interfaces/IController.sol";
import { Invoke } from "../lib/Invoke.sol";
import { ICKToken } from "../../interfaces/ICKToken.sol";
import { IWHT } from "../../interfaces/external/IWHT.sol";
import { ModuleBase } from "../lib/ModuleBase.sol";
import { Position } from "../lib/Position.sol";
import { PreciseUnitMath } from "../../lib/PreciseUnitMath.sol";
import { Uint256ArrayUtils } from "../../lib/Uint256ArrayUtils.sol";
```

```
/**
 * @title SingleIndexModule
 * @author Cook Finance
 *
 * Smart contract that facilitates rebalances for indices. Manager can set target unit amounts, max trade sizes, the
 * exchange to trade on, and the cool down period between trades (on a per asset basis). As currently constructed
 * the module only works for one Set at a time.
 *
 * SECURITY ASSUMPTION:
 *  - Works with following modules: StreamingFeeModule, BasicIssuanceModule (any other module additions to Sets using
       this module need to be examined separately)
 */
contract SingleIndexModule is ModuleBase, ReentrancyGuard {
    using SafeCast for int256;
    using SafeCast for uint256;
    using SafeMath for uint256;
    using Position for uint256;
    using Math for uint256;
    using Position for ICKToken;
    using Invoke for ICKToken;
    using AddressArrayUtils for address[];
    using Uint256ArrayUtils for uint256[];

    /* ============ Structs ============ */

    struct AssetTradeInfo {
        uint256 targetUnit;              // Target unit for the asset during current rebalance period
        uint256 maxSize;                 // Max trade size in precise units
        uint256 coolOffPeriod;           // Required time between trades for the asset
        uint256 lastTradeTimestamp;      // Timestamp of last trade
        uint256 exchange;                // Integer representing ID of exchange to use
    }

    /* ============ Enums ============ */

    // Enum of exchange Ids
    enum ExchangeId {
        None,
        Uniswap,
        Sushiswap,
```

```
    Last
}

/* ============ Events ============ */

event TargetUnitsUpdated(address indexed _component, uint256 _newUnit, uint256 _positionMultiplier);
event TradeMaximumUpdated(address indexed _component, uint256 _newMaximum);
event AssetExchangeUpdated(address indexed _component, uint256 _newExchange);
event CoolOffPeriodUpdated(address indexed _component, uint256 _newCoolOffPeriod);
event TraderStatusUpdated(address indexed _trader, bool _status);
event AnyoneTradeUpdated(bool indexed _status);
event TradeExecuted(
    address indexed _executor,
    address indexed _sellComponent,
    address indexed _buyComponent,
    uint256 _amountSold,
    uint256 _amountBought
);

/* ============ Constants ============ */

uint256 private constant TARGET_RAISE_DIVISOR = 1.0025e18;        // Raise targets 25 bps

string private constant UNISWAP_OUT = "swapTokensForExactTokens(uint256,uint256,address[],address,uint256)";
string private constant UNISWAP_IN = "swapExactTokensForTokens(uint256,uint256,address[],address,uint256)";

/* ============ State Variables ============ */

mapping(address => AssetTradeInfo) public assetInfo;     // Mapping of component to component restrictions
address[] public rebalanceComponents;                    // Components having units updated during current rebalance
uint256 public positionMultiplier;                       // Position multiplier when current rebalance units were devised
mapping(address => bool) public tradeAllowList;          // Mapping of addresses allowed to call trade()
bool public anyoneTrade;                                 // Toggles on or off skipping the tradeAllowList
ICKToken public index;                                   // Index being managed with contract
IWHT public wht;                                         // WHT contract address
address public uniswapRouter;                            // Uniswap router address
address public sushiswapRouter;                          // Sushiswap router address

/* ============ Modifiers ============ */
```

```
    modifier onlyAllowedTrader(address _caller) {
        require(_isAllowedTrader(_caller), "Address not permitted to trade");
        _;
    }

    modifier onlyEOA() {
        require(msg.sender == tx.origin, "Caller must be EOA Address");
        _;
    }

    /* ============ Constructor ============ */

    constructor(
        IController _controller,
        IWHT _wht,
        address _uniswapRouter,
        address _sushiswapRouter
    )
        public
        ModuleBase(_controller)
    {
        wht = _wht;
        uniswapRouter = _uniswapRouter;
        sushiswapRouter = _sushiswapRouter;
    }

    /**
     * MANAGER ONLY: Set new target units, zeroing out any units for components being removed from index. Log position multiplier to
     * adjust target units in case fees are accrued. Validate that WHT is not a part of the new allocation and that all components
     * in current allocation are in _components array.
     *
     * @param _newComponents                Array of new components to add to allocation
     * @param _newComponentsTargetUnits     Array of target units at end of rebalance for new components, maps to same index of component
     * @param _oldComponentsTargetUnits     Array of target units at end of rebalance for old component, maps to same index of component,
     *                                      if component being removed set to 0.
```

```
     * @param _positionMultiplier           Position multiplier when target units were calculated, needed in order to adjust target units
     *                                      if fees accrued
     */
    function startRebalance(
        address[] calldata _newComponents,
        uint256[] calldata _newComponentsTargetUnits,
        uint256[] calldata _oldComponentsTargetUnits,
        uint256 _positionMultiplier
    )
        external
        onlyManagerAndValidCK(index)
    {
        // Don't use validate arrays because empty arrays are valid
        require(_newComponents.length == _newComponentsTargetUnits.length, "Array length mismatch");

        address[] memory currentComponents = index.getComponents();
        require(currentComponents.length == _oldComponentsTargetUnits.length, "New allocation must have target for all old components");

        address[] memory aggregateComponents = currentComponents.extend(_newComponents);
        uint256[] memory aggregateTargetUnits = _oldComponentsTargetUnits.extend(_newComponentsTargetUnits);

        require(!aggregateComponents.hasDuplicate(), "Cannot duplicate components");

        for (uint256 i = 0; i < aggregateComponents.length; i++) {
            address component = aggregateComponents[i];
            uint256 targetUnit = aggregateTargetUnits[i];

            require(address(component) != address(wht), "WHT cannot be an index component");
            assetInfo[component].targetUnit = targetUnit;

            emit TargetUnitsUpdated(component, targetUnit, _positionMultiplier);
        }

        rebalanceComponents = aggregateComponents;
        positionMultiplier = _positionMultiplier;
    }

    /**
```

```
    * ACCESS LIMITED: Only approved addresses can call if anyoneTrade is false. Determines trade size
    * and direction and swaps into or out of WHT on exchange specified by manager.
    *
    * @param _component             Component to trade
    */
    function trade(address _component) external nonReentrant onlyAllowedTrader(msg.sender) onlyEOA() virtual {

        _validateTradeParameters(_component);

        (
            bool isBuy,
            uint256 tradeAmount
        ) = _calculateTradeSizeAndDirection(_component);

        if (isBuy) {
            _buyUnderweight(_component, tradeAmount);
        } else {
            _sellOverweight(_component, tradeAmount);
        }

        assetInfo[_component].lastTradeTimestamp = block.timestamp;
    }

    /**
    * ACCESS LIMITED: Only approved addresses can call if anyoneTrade is false. Only callable when 1) there are no
    * more components to be sold and, 2) entire remaining WHT amount can be traded such that resulting inflows won't
    * exceed components maxTradeSize nor overshoot the target unit. To be used near the end of rebalances when a
    * component's calculated trade size is greater in value than remaining WHT.
    *
    * @param _component             Component to trade
    */
    function tradeRemainingWHT(address _component) external nonReentrant onlyAllowedTrader(msg.sender) onlyEOA()
virtual {
        require(_noTokensToSell(), "Must sell all sellable tokens before can be called");

        _validateTradeParameters(_component);

        (, uint256 tradeLimit) = _calculateTradeSizeAndDirection(_component);

        uint256 preTradeComponentAmount = IERC20(_component).balanceOf(address(index));
```

```
        uint256 preTradeWHTAmount = wht.balanceOf(address(index));

        _executeTrade(address(wht), _component, true, preTradeWHTAmount, assetInfo[_component].exchange);

        (,
            uint256 componentTradeSize
        ) = _updatePositionState(address(wht), _component, preTradeWHTAmount, preTradeComponentAmount);

        require(componentTradeSize < tradeLimit, "Trade size exceeds trade size limit");

        assetInfo[_component].lastTradeTimestamp = block.timestamp;
    }

    /**
     * ACCESS LIMITED: For situation where all target units met and remaining WHT, uniformly raise targets by same
     * percentage in order to allow further trading. Can be called multiple times if necessary, increase should be
     * small in order to reduce tracking error.
     */
    function raiseAssetTargets() external nonReentrant onlyAllowedTrader(msg.sender) virtual {
        require(
            _allTargetsMet() && index.getDefaultPositionRealUnit(address(wht)) > 0,
            "Targets must be met and HT remaining in order to raise target"
        );

        positionMultiplier = positionMultiplier.preciseDiv(TARGET_RAISE_DIVISOR);
    }

    /**
     * MANAGER ONLY: Set trade maximums for passed components
     *
     * @param _components               Array of components
     * @param _tradeMaximums            Array of trade maximums mapping to correct component
     */
    function setTradeMaximums(
        address[] calldata _components,
        uint256[] calldata _tradeMaximums
    )
        external
        onlyManagerAndValidCK(index)
    {
```

```
        _validateArrays(_components, _tradeMaximums);

        for (uint256 i = 0; i < _components.length; i++) {
            assetInfo[_components[i]].maxSize = _tradeMaximums[i];
            emit TradeMaximumUpdated(_components[i], _tradeMaximums[i]);
        }
    }

    /**
     * MANAGER ONLY: Set exchange for passed components
     *
     * @param _components          Array of components
     * @param _exchanges           Array of exchanges mapping to correct component, uint256 used to signify exchange
     */
    function setExchanges(
        address[] calldata _components,
        uint256[] calldata _exchanges
    )
        external
        onlyManagerAndValidCK(index)
    {
        _validateArrays(_components, _exchanges);

        for (uint256 i = 0; i < _components.length; i++) {
            uint256 exchange = _exchanges[i];
            require(exchange < uint256(ExchangeId.Last), "Unrecognized exchange identifier");
            assetInfo[_components[i]].exchange = _exchanges[i];

            emit AssetExchangeUpdated(_components[i], exchange);
        }
    }

    /**
     * MANAGER ONLY: Set exchange for passed components
     *
     * @param _components             Array of components
     * @param _coolOffPeriods         Array of cool off periods to correct component
     */
    function setCoolOffPeriods(
        address[] calldata _components,
```

```
        uint256[] calldata _coolOffPeriods
    )
        external
        onlyManagerAndValidCK(index)
    {
        _validateArrays(_components, _coolOffPeriods);

        for (uint256 i = 0; i < _components.length; i++) {
            assetInfo[_components[i]].coolOffPeriod = _coolOffPeriods[i];
            emit CoolOffPeriodUpdated(_components[i], _coolOffPeriods[i]);
        }
    }

    /**
     * MANAGER ONLY: Toggle ability for passed addresses to trade from current state
     *
     * @param _traders             Array trader addresses to toggle status
     * @param _statuses            Booleans indicating if matching trader can trade
     */
    function updateTraderStatus(address[] calldata _traders, bool[] calldata _statuses) external
onlyManagerAndValidCK(index) {
        require(_traders.length == _statuses.length, "Array length mismatch");
        require(_traders.length > 0, "Array length must be > 0");
        require(!_traders.hasDuplicate(), "Cannot duplicate traders");

        for (uint256 i = 0; i < _traders.length; i++) {
            address trader = _traders[i];
            bool status = _statuses[i];
            tradeAllowList[trader] = status;
            emit TraderStatusUpdated(trader, status);
        }
    }

    /**
     * MANAGER ONLY: Toggle whether anyone can trade, bypassing the traderAllowList
     *
     * @param _status              Boolean indicating if anyone can trade
     */
    function updateAnyoneTrade(bool _status) external onlyManagerAndValidCK(index) {
        anyoneTrade = _status;
```

```
        emit AnyoneTradeUpdated(_status);
    }

    /**
     * MANAGER ONLY: Set target units to current units and last trade to zero. Initialize module.
     *
     * @param _index              Address of index being used for this CK
     */
    function initialize(ICKToken _index)
        external
        onlyCKManager(_index, msg.sender)
        onlyValidAndPendingCK(_index)
    {
        require(address(index) == address(0), "Module already in use");

        ICKToken.Position[] memory positions = _index.getPositions();

        for (uint256 i = 0; i < positions.length; i++) {
            ICKToken.Position memory position = positions[i];
            assetInfo[position.component].targetUnit = position.unit.toUint256();
            assetInfo[position.component].lastTradeTimestamp = 0;
        }

        index = _index;
    //SlowMist// The module can not be initialized again without removing the module first
        _index.initializeModule();
    }

    function removeModule() external override {}

    /* ============ Getter Functions ============ */

    /**
     * Get target units for passed components, normalized to current positionMultiplier.
     *
     * @param _components             Array of components to get target units for
     * @return                        Array of targetUnits mapping to passed components
     */
    function getTargetUnits(address[] calldata _components) external view returns(uint256[] memory) {
```

```
        uint256 currentPositionMultiplier = index.positionMultiplier().toUint256();

        uint256[] memory targetUnits = new uint256[](_components.length);
        for (uint256 i = 0; i < _components.length; i++) {
            targetUnits[i] = _normalizeTargetUnit(_components[i], currentPositionMultiplier);
        }

        return targetUnits;
    }

    function getRebalanceComponents() external view returns(address[] memory) {
        return rebalanceComponents;
    }

    /* ============ Internal Functions ============ */

    /**
     * Validate that enough time has elapsed since component's last trade and component isn't WHT.
     */
    function _validateTradeParameters(address _component) internal view virtual {
        require(rebalanceComponents.contains(_component), "Passed component not included in rebalance");

        AssetTradeInfo memory componentInfo = assetInfo[_component];
        require(componentInfo.exchange != uint256(ExchangeId.None), "Exchange must be specified");
        require(
            componentInfo.lastTradeTimestamp.add(componentInfo.coolOffPeriod) <= block.timestamp,
            "Cool off period has not elapsed."
        );
    }

    /**
     * Calculate trade size and whether trade is buy. Trade size is the minimum of the max size and components left to trade.
     * Reverts if target quantity is already met. Target unit is adjusted based on ratio of position multiplier when target was defined
     * and the current positionMultiplier.
     */
    function _calculateTradeSizeAndDirection(address _component) internal view returns (bool isBuy, uint256) {
        uint256 totalSupply = index.totalSupply();

        uint256 componentMaxSize = assetInfo[_component].maxSize;
```

```
        uint256 currentPositionMultiplier = index.positionMultiplier().toUint256();

        uint256 currentUnit = index.getDefaultPositionRealUnit(_component).toUint256();
        uint256 targetUnit = _normalizeTargetUnit(_component, currentPositionMultiplier);

        require(currentUnit != targetUnit, "Target already met");

        uint256 currentNotional = totalSupply.getDefaultTotalNotional(currentUnit);
        uint256 targetNotional = totalSupply.preciseMulCeil(targetUnit);

        return targetNotional > currentNotional ? (true, componentMaxSize.min(targetNotional.sub(currentNotional))) :
            (false, componentMaxSize.min(currentNotional.sub(targetNotional)));
    }

    /**
     * Buy an underweight asset by selling an unfixed amount of WHT for a fixed amount of the component.
     */
    function _buyUnderweight(address _component, uint256 _amount) internal {
        uint256 preTradeBuyComponentAmount = IERC20(_component).balanceOf(address(index));
        uint256 preTradeSellComponentAmount = wht.balanceOf(address(index));

        _executeTrade(address(wht), _component, false, _amount, assetInfo[_component].exchange);

        _updatePositionState(address(wht), _component, preTradeSellComponentAmount, preTradeBuyComponentAmount);
    }

    /**
     * Sell an overweight asset by selling a fixed amount of component for an unfixed amount of WHT.
     */
    function _sellOverweight(address _component, uint256 _amount) internal {
        uint256 preTradeBuyComponentAmount = wht.balanceOf(address(index));
        uint256 preTradeSellComponentAmount = IERC20(_component).balanceOf(address(index));

        _executeTrade(_component, address(wht), true, _amount, assetInfo[_component].exchange);

        _updatePositionState(_component, address(wht), preTradeSellComponentAmount, preTradeBuyComponentAmount);
    }

    /**
     * Determine parameters for trade and invoke trade on index using correct exchange.
```

```
     */
    function _executeTrade(
        address _sellComponent,
        address _buyComponent,
        bool _fixIn,
        uint256 _amount,
        uint256 _exchange
    )
        internal
        virtual
    {
        uint256 whtBalance = wht.balanceOf(address(index));

        (
            address exchangeAddress,
            bytes memory tradeCallData
        ) = _getUniswapLikeTradeData(_sellComponent, _buyComponent, _fixIn, _amount, _exchange);

        uint256 approveAmount = _sellComponent == address(wht) ? whtBalance : _amount;
        index.invokeApprove(_sellComponent, exchangeAddress, approveAmount);
        index.invoke(exchangeAddress, 0, tradeCallData);
    }

    /**
     * Update position units on index. Emit event.
     */
    function _updatePositionState(
        address _sellComponent,
        address _buyComponent,
        uint256 _preTradeSellComponentAmount,
        uint256 _preTradeBuyComponentAmount
    )
        internal
        returns (uint256 sellAmount, uint256 buyAmount)
    {
        uint256 totalSupply = index.totalSupply();

        (uint256 postTradeSellComponentAmount,,) = index.calculateAndEditDefaultPosition(
            _sellComponent,
            totalSupply,
```

```
            _preTradeSellComponentAmount
        );
        (uint256 postTradeBuyComponentAmount,,) = index.calculateAndEditDefaultPosition(
            _buyComponent,
            totalSupply,
            _preTradeBuyComponentAmount
        );

        sellAmount = _preTradeSellComponentAmount.sub(postTradeSellComponentAmount);
        buyAmount = postTradeBuyComponentAmount.sub(_preTradeBuyComponentAmount);

        emit TradeExecuted(
            msg.sender,
            _sellComponent,
            _buyComponent,
            sellAmount,
            buyAmount
        );
    }

    /**
     * Determine whether exchange to call is Uniswap or Sushiswap and generate necessary call data.
     */
    function _getUniswapLikeTradeData(
        address _sellComponent,
        address _buyComponent,
        bool _fixIn,
        uint256 _amount,
        uint256 _exchange
    )
        internal
        view
        returns(address, bytes memory)
    {
        address exchangeAddress = _exchange == uint256(ExchangeId.Uniswap) ? uniswapRouter : sushiswapRouter;

        string memory functionSignature;
        address[] memory path = new address[](2);
        uint256 limit;
        if (_fixIn) {
```

```
            functionSignature = UNISWAP_IN;
            limit = 1;
        } else {
            functionSignature = UNISWAP_OUT;
            limit = PreciseUnitMath.maxUint256();
        }
        path[0] = _sellComponent;
        path[1] = _buyComponent;
```

//SlowMist// The swap data construct here does not set the corrsponding limitation, when trade, it may suffer from a slippage issue.

```
        bytes memory tradeCallData = abi.encodeWithSignature(
            functionSignature,
            _amount,
            limit,
            path,
            address(index),
            now.add(180)
        );

        return (exchangeAddress, tradeCallData);
    }

    /**
     * Check if there are any more tokens to sell.
     */
    function _noTokensToSell() internal view returns (bool) {
        uint256 currentPositionMultiplier = index.positionMultiplier().toUint256();
        for (uint256 i = 0; i < rebalanceComponents.length; i++) {
            address component = rebalanceComponents[i];
            bool canSell = _normalizeTargetUnit(component, currentPositionMultiplier) < index.getDefaultPositionRealUnit(
                component
            ).toUint256();
            if (canSell) { return false; }
        }
        return true;
    }

    /**
     * Check if all targets are met
```

```
     */
    function _allTargetsMet() internal view returns (bool) {
        uint256 currentPositionMultiplier = index.positionMultiplier().toUint256();
        for (uint256 i = 0; i < rebalanceComponents.length; i++) {
            address component = rebalanceComponents[i];
            bool targetUnmet = _normalizeTargetUnit(component, currentPositionMultiplier) !=
index.getDefaultPositionRealUnit(
                    component
            ).toUint256();
            if (targetUnmet) { return false; }
        }
        return true;
    }

    /**
     * Normalize target unit to current position multiplier in case fees have been accrued.
     */
    function _normalizeTargetUnit(address _component, uint256 _currentPositionMultiplier) internal view returns(uint256) {
        return assetInfo[_component].targetUnit.mul(_currentPositionMultiplier).div(positionMultiplier);
    }

    /**
     * Determine if passed address is allowed to call trade. If anyoneTrade set to true anyone can call otherwise needs to be
approved.
     */
    function _isAllowedTrader(address _caller) internal view virtual returns (bool) {
        return anyoneTrade || tradeAllowList[_caller];
    }

    /**
     * Validate arrays are of equal length and not empty.
     */
    function _validateArrays(address[] calldata _components, uint256[] calldata _data) internal pure {
        require(_components.length == _data.length, "Array length mismatch");
        require(_components.length > 0, "Array length must be > 0");
        require(!_components.hasDuplicate(), "Cannot duplicate components");
    }
}
```

TradeModule.sol

```
/*
    Copyright 2021 Cook Finance.

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.

    SPDX-License-Identifier: Apache License, Version 2.0
*/

pragma solidity ^0.6.10;
pragma experimental "ABIEncoderV2";

import { ReentrancyGuard } from "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import { SafeMath } from "@openzeppelin/contracts/math/SafeMath.sol";
import { SafeCast } from "@openzeppelin/contracts/utils/SafeCast.sol";

import { IController } from "../../interfaces/IController.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { IExchangeAdapter } from "../../interfaces/IExchangeAdapter.sol";
import { IIntegrationRegistry } from "../../interfaces/IIntegrationRegistry.sol";
import { Invoke } from "../lib/Invoke.sol";
import { ICKToken } from "../../interfaces/ICKToken.sol";
import { ModuleBase } from "../lib/ModuleBase.sol";
import { Position } from "../lib/Position.sol";
import { PreciseUnitMath } from "../../lib/PreciseUnitMath.sol";

/**
 * @title TradeModule
 * @author Cook Finance
 *
 * Module that enables CKTokens to perform atomic trades using Decentralized Exchanges
```

```
 * such as Uniswap. Integrations mappings are stored on the IntegrationRegistry contract.
 */
contract TradeModule is ModuleBase, ReentrancyGuard {
    using SafeCast for int256;
    using SafeMath for uint256;

    using Invoke for ICKToken;
    using Position for ICKToken;
    using PreciseUnitMath for uint256;

    /* ============ Struct ============ */

    struct TradeInfo {
        ICKToken ckToken;                       // Instance of CKToken
        IExchangeAdapter exchangeAdapter;           // Instance of exchange adapter contract
        address sendToken;                      // Address of token being sold
        address receiveToken;                   // Address of token being bought
        uint256 ckTotalSupply;                  // Total supply of CKToken in Precise Units (10^18)
        uint256 totalSendQuantity;              // Total quantity of sold token (position unit x total supply)
        uint256 totalMinReceiveQuantity;        // Total minimum quantity of token to receive back
        uint256 preTradeSendTokenBalance;       // Total initial balance of token being sold
        uint256 preTradeReceiveTokenBalance;    // Total initial balance of token being bought
    }

    /* ============ Events ============ */

    event ComponentExchanged(
        ICKToken indexed _ckToken,
        address indexed _sendToken,
        address indexed _receiveToken,
        IExchangeAdapter _exchangeAdapter,
        uint256 _totalSendAmount,
        uint256 _totalReceiveAmount,
        uint256 _protocolFee
    );

    /* ============ Constants ============ */

    // 0 index stores the fee % charged in the trade function
    uint256 constant internal TRADE_MODULE_PROTOCOL_FEE_INDEX = 0;
```

```
/* ============ Constructor ============ */

constructor(IController _controller) public ModuleBase(_controller) {}

/* ============ External Functions ============ */

/**
 * Initializes this module to the CKToken. Only callable by the CKToken's manager.
 *
 * @param _ckToken                  Instance of the CKToken to initialize
 */
function initialize(
    ICKToken _ckToken
)
    external
    onlyValidAndPendingCK(_ckToken)
    onlyCKManager(_ckToken, msg.sender)
{
  //SlowMist// The module can not be initialized again without removing the module first
    _ckToken.initializeModule();
}

/**
 * Executes a trade on a supported DEX. Only callable by the CKToken's manager.
 * @dev Although the CKToken units are passed in for the send and receive quantities, the total quantity
 * sent and received is the quantity of CKToken units multiplied by the CKToken totalSupply.
 *
 * @param _ckToken              Instance of the CKToken to trade
 * @param _exchangeName         Human readable name of the exchange in the integrations registry
 * @param _sendToken            Address of the token to be sent to the exchange
 * @param _sendQuantity         Units of token in CKToken sent to the exchange
 * @param _receiveToken         Address of the token that will be received from the exchange
 * @param _minReceiveQuantity   Min units of token in CKToken to be received from the exchange
 * @param _data                 Arbitrary bytes to be used to construct trade call data
 */
function trade(
    ICKToken _ckToken,
    string memory _exchangeName,
```

```
        address _sendToken,
        uint256 _sendQuantity,
        address _receiveToken,
        uint256 _minReceiveQuantity,
        bytes memory _data
    )
        external
        nonReentrant
        onlyManagerAndValidCK(_ckToken)
    {
        TradeInfo memory tradeInfo = _createTradeInfo(
            _ckToken,
            _exchangeName,
            _sendToken,
            _receiveToken, //SlowMist// The address is up the stack with manager contract
            _sendQuantity,
            _minReceiveQuantity
        );

        _validatePreTradeData(tradeInfo, _sendQuantity);

        _executeTrade(tradeInfo, _data);

        uint256 exchangedQuantity = _validatePostTrade(tradeInfo);

        uint256 protocolFee = _accrueProtocolFee(tradeInfo, exchangedQuantity);

        (
            uint256 netSendAmount,
            uint256 netReceiveAmount
        ) = _updateCKTokenPositions(tradeInfo);

        emit ComponentExchanged(
            _ckToken,
            _sendToken,
            _receiveToken,
            tradeInfo.exchangeAdapter,
            netSendAmount,
            netReceiveAmount,
```

```
            protocolFee
        );
    }

    /**
     * Removes this module from the CKToken, via call by the CKToken. Left with empty logic
     * here because there are no check needed to verify removal.
     */
    function removeModule() external override {}

    /* ============ Internal Functions ============ */

    /**
     * Create and return TradeInfo struct
     *
     * @param _ckToken               Instance of the CKToken to trade
     * @param _exchangeName          Human readable name of the exchange in the integrations registry
     * @param _sendToken             Address of the token to be sent to the exchange
     * @param _receiveToken          Address of the token that will be received from the exchange
     * @param _sendQuantity          Units of token in CKToken sent to the exchange
     * @param _minReceiveQuantity    Min units of token in CKToken to be received from the exchange
     *
     * return TradeInfo              Struct containing data for trade
     */
    function _createTradeInfo(
        ICKToken _ckToken,
        string memory _exchangeName,
        address _sendToken,
        address _receiveToken,
        uint256 _sendQuantity,
        uint256 _minReceiveQuantity
    )
        internal
        view
        returns (TradeInfo memory)
    {
        TradeInfo memory tradeInfo;

        tradeInfo.ckToken = _ckToken;
```

```
        tradeInfo.exchangeAdapter = IExchangeAdapter(getAndValidateAdapter(_exchangeName));

        tradeInfo.sendToken = _sendToken;
        tradeInfo.receiveToken = _receiveToken;

        tradeInfo.ckTotalSupply = _ckToken.totalSupply();

        tradeInfo.totalSendQuantity = Position.getDefaultTotalNotional(tradeInfo.ckTotalSupply, _sendQuantity);

        tradeInfo.totalMinReceiveQuantity = Position.getDefaultTotalNotional(tradeInfo.ckTotalSupply, _minReceiveQuantity);

        tradeInfo.preTradeSendTokenBalance = IERC20(_sendToken).balanceOf(address(_ckToken));
        tradeInfo.preTradeReceiveTokenBalance = IERC20(_receiveToken).balanceOf(address(_ckToken));

        return tradeInfo;
    }

    /**
     * Validate pre trade data. Check exchange is valid, token quantity is valid.
     *
     * @param _tradeInfo              Struct containing trade information used in internal functions
     * @param _sendQuantity           Units of token in CKToken sent to the exchange
     */
    function _validatePreTradeData(TradeInfo memory _tradeInfo, uint256 _sendQuantity) internal view {
        require(_tradeInfo.totalSendQuantity > 0, "Token to sell must be nonzero");

        require(
            _tradeInfo.ckToken.hasSufficientDefaultUnits(_tradeInfo.sendToken, _sendQuantity),
            "Unit cant be greater than existing"
        );
    }

    /**
     * Invoke approve for send token, get method data and invoke trade in the context of the CKToken.
     *
     * @param _tradeInfo              Struct containing trade information used in internal functions
     * @param _data                   Arbitrary bytes to be used to construct trade call data
     */
    function _executeTrade(
        TradeInfo memory _tradeInfo,
```

```
        bytes memory _data
    )
        internal
    {
        // Get spender address from exchange adapter and invoke approve for exact amount on CKToken
        _tradeInfo.ckToken.invokeApprove(
            _tradeInfo.sendToken,
            _tradeInfo.exchangeAdapter.getSpender(),
            _tradeInfo.totalSendQuantity
        );

        (
            address targetExchange,
            uint256 callValue,
            bytes memory methodData
        ) = _tradeInfo.exchangeAdapter.getTradeCalldata(
            _tradeInfo.sendToken,
            _tradeInfo.receiveToken,
            address(_tradeInfo.ckToken),
            _tradeInfo.totalSendQuantity,
            _tradeInfo.totalMinReceiveQuantity,
            _data
        );

        _tradeInfo.ckToken.invoke(targetExchange, callValue, methodData);
    }

    /**
     * Validate post trade data.
     *
     * @param _tradeInfo                Struct containing trade information used in internal functions
     * @return uint256                  Total quantity of receive token that was exchanged
     */
    function _validatePostTrade(TradeInfo memory _tradeInfo) internal view returns (uint256) {
        uint256 exchangedQuantity = IERC20(_tradeInfo.receiveToken)
            .balanceOf(address(_tradeInfo.ckToken))
            .sub(_tradeInfo.preTradeReceiveTokenBalance);

        require(
            exchangedQuantity >= _tradeInfo.totalMinReceiveQuantity,
```

```
            "Slippage greater than allowed"
        );

        return exchangedQuantity;
    }

    /**
     * Retrieve fee from controller and calculate total protocol fee and send from CKToken to protocol recipient
     *
     * @param _tradeInfo                Struct containing trade information used in internal functions
     * @return uint256                  Amount of receive token taken as protocol fee
     */
    function _accrueProtocolFee(TradeInfo memory _tradeInfo, uint256 _exchangedQuantity) internal returns (uint256) {
        uint256 protocolFeeTotal = getModuleFee(TRADE_MODULE_PROTOCOL_FEE_INDEX, _exchangedQuantity);

        payProtocolFeeFromCKToken(_tradeInfo.ckToken, _tradeInfo.receiveToken, protocolFeeTotal);

        return protocolFeeTotal;
    }

    /**
     * Update CKToken positions
     *
     * @param _tradeInfo                Struct containing trade information used in internal functions
     * @return uint256                  Amount of sendTokens used in the trade
     * @return uint256                  Amount of receiveTokens received in the trade (net of fees)
     */
    function _updateCKTokenPositions(TradeInfo memory _tradeInfo) internal returns (uint256, uint256) {
        (uint256 currentSendTokenBalance,,) = _tradeInfo.ckToken.calculateAndEditDefaultPosition(
            _tradeInfo.sendToken,
            _tradeInfo.ckTotalSupply,
            _tradeInfo.preTradeSendTokenBalance
        );

        (uint256 currentReceiveTokenBalance,,) = _tradeInfo.ckToken.calculateAndEditDefaultPosition(
            _tradeInfo.receiveToken,
            _tradeInfo.ckTotalSupply,
            _tradeInfo.preTradeReceiveTokenBalance
        );
```

```
        return (
            _tradeInfo.preTradeSendTokenBalance.sub(currentSendTokenBalance),
            currentReceiveTokenBalance.sub(_tradeInfo.preTradeReceiveTokenBalance)
        );
    }
}
```

WarpModule.sol

```
/*
    Copyright 2021 Cook Finance.

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.

    SPDX-License-Identifier: Apache License, Version 2.0
*/

pragma solidity 0.6.10;
pragma experimental "ABIEncoderV2";

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { ReentrancyGuard } from "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import { SafeCast } from "@openzeppelin/contracts/utils/SafeCast.sol";
import { SafeMath } from "@openzeppelin/contracts/math/SafeMath.sol";

import { IController } from "../../interfaces/IController.sol";
import { IIntegrationRegistry } from "../../interfaces/IIntegrationRegistry.sol";
import { Invoke } from "../lib/Invoke.sol";
import { ICKToken } from "../../interfaces/ICKToken.sol";
import { IWHT } from "../../interfaces/external/IWHT.sol";
import { IWrapAdapter } from "../../interfaces/IWrapAdapter.sol";
```

```
import { ModuleBase } from "../lib/ModuleBase.sol";
import { Position } from "../lib/Position.sol";
import { PreciseUnitMath } from "../../lib/PreciseUnitMath.sol";

/**
 * @title WrapModule
 * @author Cook Finance
 *
 * Module that enables the wrapping of ERC20 and HT positions via third party protocols. The WrapModule
 * works in conjunction with WrapAdapters, in which the wrapAdapterID / integrationNames are stored on the
 * integration registry.
 *
 * Some examples of wrap actions include wrapping, DAI to cDAI (Compound) or Dai to aDai (AAVE).
 */
contract WrapModule is ModuleBase, ReentrancyGuard {
    using SafeCast for int256;
    using PreciseUnitMath for uint256;
    using Position for uint256;
    using SafeMath for uint256;

    using Invoke for ICKToken;
    using Position for ICKToken.Position;
    using Position for ICKToken;

    /* ============ Events ============ */

    event ComponentWrapped(
        ICKToken indexed _ckToken,
        address indexed _underlyingToken,
        address indexed _wrappedToken,
        uint256 _underlyingQuantity,
        uint256 _wrappedQuantity,
        string _integrationName
    );

    event ComponentUnwrapped(
        ICKToken indexed _ckToken,
        address indexed _underlyingToken,
        address indexed _wrappedToken,
        uint256 _underlyingQuantity,
```

```
        uint256 _wrappedQuantity,
        string _integrationName
    );

    /* ============ State Variables ============ */

    // Wrapped HT address
    IWHT public wht;

    /* ============ Constructor ============ */

    /**
     * @param _controller               Address of controller contract
     * @param _wht                      Address of wrapped HT
     */
    constructor(IController _controller, IWHT _wht) public ModuleBase(_controller) {
        wht = _wht;
    }

    /* ============ External Functions ============ */

    /**
     * MANAGER-ONLY: Instructs the CKToken to wrap an underlying asset into a wrappedToken via a specified adapter.
     *
     * @param _ckToken               Instance of the CKToken
     * @param _underlyingToken       Address of the component to be wrapped
     * @param _wrappedToken          Address of the desired wrapped token
     * @param _underlyingUnits       Quantity of underlying units in Position units
     * @param _integrationName       Name of wrap module integration (mapping on integration registry)
     */
    function wrap(
        ICKToken _ckToken,
        address _underlyingToken,
        address _wrappedToken,
        uint256 _underlyingUnits,
        string calldata _integrationName
    )
        external
        nonReentrant
        onlyManagerAndValidCK(_ckToken)
```

```
{
    (
        uint256 notionalUnderlyingWrapped,
        uint256 notionalWrapped
    ) = _validateWrapAndUpdate(
        _integrationName,
        _ckToken,
        _underlyingToken,
        _wrappedToken,
        _underlyingUnits,
        false // does not use HT
    );

    emit ComponentWrapped(
        _ckToken,
        _underlyingToken,
        _wrappedToken,
        notionalUnderlyingWrapped,
        notionalWrapped,
        _integrationName
    );
}

/**
 * MANAGER-ONLY: Instructs the CKToken to wrap HT into a wrappedToken via a specified adapter. Since CKTokens
 * only hold WHT, in order to support protocols that collateralize with HT the CKToken's WHT must be unwrapped
 * first before sending to the external protocol.
 *
 * @param _ckToken               Instance of the CKToken
 * @param _wrappedToken          Address of the desired wrapped token
 * @param _underlyingUnits       Quantity of underlying units in Position units
 * @param _integrationName       Name of wrap module integration (mapping on integration registry)
 */
function wrapWithHT(
    ICKToken _ckToken,
    address _wrappedToken,
    uint256 _underlyingUnits,
    string calldata _integrationName
)
    external
```

```
    nonReentrant
    onlyManagerAndValidCK(_ckToken)
{
    (
        uint256 notionalUnderlyingWrapped,
        uint256 notionalWrapped
    ) = _validateWrapAndUpdate(
        _integrationName,
        _ckToken,
        address(wht),
        _wrappedToken,
        _underlyingUnits,
        true // uses HT
    );

    emit ComponentWrapped(
        _ckToken,
        address(wht),
        _wrappedToken,
        notionalUnderlyingWrapped,
        notionalWrapped,
        _integrationName
    );
}

/**
 * MANAGER-ONLY: Instructs the CKToken to unwrap a wrapped asset into its underlying via a specified adapter.
 *
 * @param _ckToken               Instance of the CKToken
 * @param _underlyingToken       Address of the underlying asset
 * @param _wrappedToken          Address of the component to be unwrapped
 * @param _wrappedUnits          Quantity of wrapped tokens in Position units
 * @param _integrationName       ID of wrap module integration (mapping on integration registry)
 */
function unwrap(
    ICKToken _ckToken,
    address _underlyingToken,
    address _wrappedToken,
    uint256 _wrappedUnits,
    string calldata _integrationName
```

```
)
    external
    nonReentrant
    onlyManagerAndValidCK(_ckToken)
{
    (
        uint256 notionalUnderlyingUnwrapped,
        uint256 notionalUnwrapped
    ) = _validateUnwrapAndUpdate(
        _integrationName,
        _ckToken,
        _underlyingToken,
        _wrappedToken,
        _wrappedUnits,
        false // uses HT
    );

    emit ComponentUnwrapped(
        _ckToken,
        _underlyingToken,
        _wrappedToken,
        notionalUnderlyingUnwrapped,
        notionalUnwrapped,
        _integrationName
    );
}

/**
 * MANAGER-ONLY: Instructs the CKToken to unwrap a wrapped asset collateralized by HT into Wrapped HT. Since
 * external protocol will send back HT that HT must be Wrapped into WHT in order to be accounted for by CKToken.
 *
 * @param _ckToken                  Instance of the CKToken
 * @param _wrappedToken               Address of the component to be unwrapped
 * @param _wrappedUnits               Quantity of wrapped tokens in Position units
 * @param _integrationName          ID of wrap module integration (mapping on integration registry)
 */
function unwrapWithHT(
    ICKToken _ckToken,
    address _wrappedToken,
    uint256 _wrappedUnits,
```

```
        string calldata _integrationName
    )
        external
        nonReentrant
        onlyManagerAndValidCK(_ckToken)
    {
        (
            uint256 notionalUnderlyingUnwrapped,
            uint256 notionalUnwrapped
        ) = _validateUnwrapAndUpdate(
            _integrationName,
            _ckToken,
            address(wht),
            _wrappedToken,
            _wrappedUnits,
            true // uses HT
        );

        emit ComponentUnwrapped(
            _ckToken,
            address(wht),
            _wrappedToken,
            notionalUnderlyingUnwrapped,
            notionalUnwrapped,
            _integrationName
        );
    }

    /**
     * Initializes this module to the CKToken. Only callable by the CKToken's manager.
     *
     * @param _ckToken               Instance of the CKToken to issue
     */
    function initialize(ICKToken _ckToken) external onlyCKManager(_ckToken, msg.sender) {
        require(controller.isCK(address(_ckToken)), "Must be controller-enabled CKToken");
        require(isCKPendingInitialization(_ckToken), "Must be pending initialization");
```

//SlowMist// The module can not be initialized again without removing the module first.

```
        _ckToken.initializeModule();
    }
```

```
    /**
     * Removes this module from the CKToken, via call by the CKToken.
     */
    function removeModule() external override {}


    /* ============ Internal Functions ============ */

    /**
     * Validates the wrap operation is valid. In particular, the following checks are made:
     * - The position is Default
     * - The position has sufficient units given the transact quantity
     * - The transact quantity > 0
     *
     * It is expected that the adapter will check if wrappedToken/underlyingToken are a valid pair for the given
     * integration.
     */
    function _validateInputs(
        ICKToken _ckToken,
        address _transactPosition,
        uint256 _transactPositionUnits
    )
        internal
        view
    {
        require(_transactPositionUnits > 0, "Target position units must be > 0");
        require(_ckToken.hasDefaultPosition(_transactPosition), "Target default position must be component");
        require(
            _ckToken.hasSufficientDefaultUnits(_transactPosition, _transactPositionUnits),
            "Unit cant be greater than existing"
        );
    }

    /**
     * The WrapModule calculates the total notional underlying to wrap, approves the underlying to the 3rd party
     * integration contract, then invokes the CKToken to call wrap by passing its calldata along. When raw HT
     * is being used (_usesHT = true) WTH position must first be unwrapped and underlyingAddress sent to
     * adapter must be external protocol's HT representative address.
     *
```

```
     * Returns notional amount of underlying tokens and wrapped tokens that were wrapped.
     */
    function _validateWrapAndUpdate(
        string calldata _integrationName,
        ICKToken _ckToken,
        address _underlyingToken,
        address _wrappedToken,
        uint256 _underlyingUnits,
        bool _usesHT
    )
        internal
        returns (uint256, uint256)
    {
        _validateInputs(_ckToken, _underlyingToken, _underlyingUnits);

        // Snapshot pre wrap balances
        (
            uint256 preActionUnderlyingNotional,
            uint256 preActionWrapNotional
        ) = _snapshotTargetAssetsBalance(_ckToken, _underlyingToken, _wrappedToken);

        uint256 notionalUnderlying = _ckToken.totalSupply().getDefaultTotalNotional(_underlyingUnits);
        IWrapAdapter wrapAdapter = IWrapAdapter(getAndValidateAdapter(_integrationName));

        // Execute any pre-wrap actions depending on if using raw HT or not
        if (_usesHT) {
            _ckToken.invokeUnwrapWHT(address(wht), notionalUnderlying);
        } else {
            address spender = wrapAdapter.getSpenderAddress(_underlyingToken, _wrappedToken);

            _ckToken.invokeApprove(_underlyingToken, spender, notionalUnderlying);
        }

        // Get function call data and invoke on CKToken
        _createWrapDataAndInvoke(
            _ckToken,
            wrapAdapter,
            _usesHT ? wrapAdapter.HT_TOKEN_ADDRESS() : _underlyingToken,
            _wrappedToken,
            notionalUnderlying
```

```
        );

        // Snapshot post wrap balances
        (
            uint256 postActionUnderlyingNotional,
            uint256 postActionWrapNotional
        ) = _snapshotTargetAssetsBalance(_ckToken, _underlyingToken, _wrappedToken);

        _updatePosition(_ckToken, _underlyingToken, preActionUnderlyingNotional, postActionUnderlyingNotional);
        _updatePosition(_ckToken, _wrappedToken, preActionWrapNotional, postActionWrapNotional);

        return (
            preActionUnderlyingNotional.sub(postActionUnderlyingNotional),
            postActionWrapNotional.sub(preActionWrapNotional)
        );
    }

    /**
     * The WrapModule calculates the total notional wrap token to unwrap, then invokes the CKToken to call
     * unwrap by passing its calldata along. When raw HT is being used (_usesHT = true) underlyingAddress
     * sent to adapter must be set to external protocol's HT representative address and HT returned from
     * external protocol is wrapped.
     *
     * Returns notional amount of underlying tokens and wrapped tokens unwrapped.
     */
    function _validateUnwrapAndUpdate(
        string calldata _integrationName,
        ICKToken _ckToken,
        address _underlyingToken,
        address _wrappedToken,
        uint256 _wrappedTokenUnits,
        bool _usesHT
    )
        internal
        returns (uint256, uint256)
    {
        _validateInputs(_ckToken, _wrappedToken, _wrappedTokenUnits);

        (
            uint256 preActionUnderlyingNotional,
```

```
            uint256 preActionWrapNotional
        ) = _snapshotTargetAssetsBalance(_ckToken, _underlyingToken, _wrappedToken);

        uint256 notionalWrappedToken = _ckToken.totalSupply().getDefaultTotalNotional(_wrappedTokenUnits);
        IWrapAdapter wrapAdapter = IWrapAdapter(getAndValidateAdapter(_integrationName));

        // Get function call data and invoke on CKToken
        _createUnwrapDataAndInvoke(
            _ckToken,
            wrapAdapter,
            _usesHT ? wrapAdapter.HT_TOKEN_ADDRESS() : _underlyingToken,
            _wrappedToken,
            notionalWrappedToken
        );

        if (_usesHT) {
            _ckToken.invokeWrapWHT(address(wht), address(_ckToken).balance);
        }

        (
            uint256 postActionUnderlyingNotional,
            uint256 postActionWrapNotional
        ) = _snapshotTargetAssetsBalance(_ckToken, _underlyingToken, _wrappedToken);

        _updatePosition(_ckToken, _underlyingToken, preActionUnderlyingNotional, postActionUnderlyingNotional);
        _updatePosition(_ckToken, _wrappedToken, preActionWrapNotional, postActionWrapNotional);

        return (
            postActionUnderlyingNotional.sub(preActionUnderlyingNotional),
            preActionWrapNotional.sub(postActionWrapNotional)
        );
    }

    /**
     * Create the calldata for wrap and then invoke the call on the CKToken.
     */
    function _createWrapDataAndInvoke(
        ICKToken _ckToken,
        IWrapAdapter _wrapAdapter,
        address _underlyingToken,
```

```
        address _wrappedToken,
        uint256 _notionalUnderlying
    ) internal {
        (
            address callTarget,
            uint256 callValue,
            bytes memory callByteData
        ) = _wrapAdapter.getWrapCallData(
            _underlyingToken,
            _wrappedToken,
            _notionalUnderlying
        );

        _ckToken.invoke(callTarget, callValue, callByteData);
    }

    /**
     * Create the calldata for unwrap and then invoke the call on the CKToken.
     */
    function _createUnwrapDataAndInvoke(
        ICKToken _ckToken,
        IWrapAdapter _wrapAdapter,
        address _underlyingToken,
        address _wrappedToken,
        uint256 _notionalUnderlying
    ) internal {
        (
            address callTarget,
            uint256 callValue,
            bytes memory callByteData
        ) = _wrapAdapter.getUnwrapCallData(
            _underlyingToken,
            _wrappedToken,
            _notionalUnderlying
        );

        _ckToken.invoke(callTarget, callValue, callByteData);
    }

    /**
```

```
    * After a wrap/unwrap operation, check the underlying and wrap token quantities and recalculate
    * the units ((total tokens - airdrop)/ total supply). Then update the position on the CKToken.
    */
    function _updatePosition(
        ICKToken _ckToken,
        address _token,
        uint256 _preActionTokenBalance,
        uint256 _postActionTokenBalance
    ) internal {
        uint256 newUnit = _ckToken.totalSupply().calculateDefaultEditPositionUnit(
            _preActionTokenBalance,
            _postActionTokenBalance,
            _ckToken.getDefaultPositionRealUnit(_token).toUint256()
        );

        _ckToken.editDefaultPosition(_token, newUnit);
    }

    /**
    * Take snapshot of CKToken's balance of underlying and wrapped tokens.
    */
    function _snapshotTargetAssetsBalance(
        ICKToken _ckToken,
        address _underlyingToken,
        address _wrappedToken
    ) internal view returns(uint256, uint256) {
        uint256 underlyingTokenBalance = IERC20(_underlyingToken).balanceOf(address(_ckToken));
        uint256 wrapTokenBalance = IERC20(_wrappedToken).balanceOf(address(_ckToken));

        return (
            underlyingTokenBalance,
            wrapTokenBalance
        );
    }
}
```

UniswapPairPriceAdapter.sol

```
/*
    Copyright 2021 Cook Finance.
    Licensed under the Apache License, Version 2.0 (the "License");
```

```
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at
    http://www.apache.org/licenses/LICENSE-2.0
    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
    SPDX-License-Identifier: Apache License, Version 2.0
*/

pragma solidity 0.6.10;

import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import { Ownable } from "@openzeppelin/contracts/access/Ownable.sol";
import { SafeMath } from "@openzeppelin/contracts/math/SafeMath.sol";

import { AddressArrayUtils } from "../../lib/AddressArrayUtils.sol";
import { IController } from "../../interfaces/IController.sol";
import { IPriceOracle } from "../../interfaces/IPriceOracle.sol";
import { IUniswapV2Pair } from "../../interfaces/external/IUniswapV2Pair.sol";
import { UniswapV2Library } from "../../../external/contracts/uniswap/lib/UniswapV2Library.sol";
import { PreciseUnitMath } from "../../lib/PreciseUnitMath.sol";
import { ResourceIdentifier } from "../lib/ResourceIdentifier.sol";

contract UniswapPairPriceAdapter is Ownable {
    using AddressArrayUtils for address[];
    using SafeMath for uint256;
    using PreciseUnitMath for uint256;
    using ResourceIdentifier for IController;

    /* ============ Structs ============ */

    /**
     * Struct containing information for get price function
     */
    struct PoolSettings {
        address tokenOne;                   // Address of first token in reserve
        address tokenTwo;                   // Address of second token in reserve
        uint256 tokenOneBaseUnit;           // Token one base unit. E.g. ETH is 10e18, USDC is 10e6
```

```
    uint256 tokenTwoBaseUnit;              // Token two base unit.
    bool isValid;                          // Boolean that returns if Uniswap pool is allowed
}

/* ============ State Variables ============ */

// Instance of the Controller contract
IController public controller;

// Uniswap allowed pools to settings mapping
mapping(address => PoolSettings) public uniswapPoolsToSettings;

// Uniswap allowed pools
address[] public allowedUniswapPools;

// Address of Uniswap factory
address public uniswapFactory;

/* ============ Constructor ============ */

/**
 * Set state variables
 *
 * @param _controller           Instance of controller contract
 * @param _uniswapFactory       Address of Uniswap factory
 * @param _uniswapPools         Array of allowed Uniswap pools
 */
constructor(
    IController _controller,
    address _uniswapFactory,
    IUniswapV2Pair[] memory _uniswapPools
)
    public
{
    controller = _controller;
    uniswapFactory = _uniswapFactory;

    // Add each of initial addresses to state
    for (uint256 i = 0; i < _uniswapPools.length; i++) {
        IUniswapV2Pair uniswapPoolToAdd = _uniswapPools[i];
```

```
        // Require pools are unique
        require(
            !uniswapPoolsToSettings[address(uniswapPoolToAdd)].isValid,
            "Uniswap pool address must be unique."
        );

        // Initialize pool settings
        PoolSettings memory poolSettings;
        poolSettings.tokenOne = uniswapPoolToAdd.token0();
        poolSettings.tokenTwo = uniswapPoolToAdd.token1();
        uint256 tokenOneDecimals = ERC20(poolSettings.tokenOne).decimals();
        poolSettings.tokenOneBaseUnit = 10 ** tokenOneDecimals;
        uint256 tokenTwoDecimals = ERC20(poolSettings.tokenTwo).decimals();
        poolSettings.tokenTwoBaseUnit = 10 ** tokenTwoDecimals;
        poolSettings.isValid = true;

        // Add to storage
        allowedUniswapPools.push(address(uniswapPoolToAdd));
        uniswapPoolsToSettings[address(uniswapPoolToAdd)] = poolSettings;
    }
}

/* ============ External Functions ============ */

/**
 * Calculate price from Uniswap. Note: must be system contract to be able to retrieve price. If both assets are
 * not Uniswap pool, return false.
 *
 * @param _assetOne          Address of first asset in pair
 * @param _assetTwo          Address of second asset in pair
 */
function getPrice(address _assetOne, address _assetTwo) external view returns (bool, uint256) {
    require(controller.isSystemContract(msg.sender), "Must be system contract");

    bool isAllowedUniswapPoolOne = uniswapPoolsToSettings[_assetOne].isValid;
    bool isAllowedUniswapPoolTwo = uniswapPoolsToSettings[_assetTwo].isValid;

    // If assetOne and assetTwo are both not Uniswap pools, then return false
    if (!isAllowedUniswapPoolOne && !isAllowedUniswapPoolTwo) {
```

```
            return (false, 0);
        }

        IPriceOracle priceOracle = controller.getPriceOracle();
        address masterQuoteAsset = priceOracle.masterQuoteAsset();

        uint256 assetOnePriceToMaster;
        if(isAllowedUniswapPoolOne) {
            assetOnePriceToMaster = _getUniswapPrice(priceOracle, _assetOne, masterQuoteAsset);
        } else {
            assetOnePriceToMaster = priceOracle.getPrice(_assetOne, masterQuoteAsset);
        }

        uint256 assetTwoPriceToMaster;
        if(isAllowedUniswapPoolTwo) {
            assetTwoPriceToMaster = _getUniswapPrice(priceOracle, _assetTwo, masterQuoteAsset);
        } else {
            assetTwoPriceToMaster = priceOracle.getPrice(_assetTwo, masterQuoteAsset);
        }

        return (true, assetOnePriceToMaster.preciseDiv(assetTwoPriceToMaster));
    }

    function addPool(address _poolAddress) external onlyOwner {
        require (
            !uniswapPoolsToSettings[_poolAddress].isValid,
            "Uniswap pool address already added"
        );
        IUniswapV2Pair poolToken = IUniswapV2Pair(_poolAddress);

        uniswapPoolsToSettings[_poolAddress].tokenOne = poolToken.token0();
        uniswapPoolsToSettings[_poolAddress].tokenTwo = poolToken.token1();
        uint256 tokenOneDecimals = ERC20(uniswapPoolsToSettings[_poolAddress].tokenOne).decimals();
        uniswapPoolsToSettings[_poolAddress].tokenOneBaseUnit = 10 ** tokenOneDecimals;
        uint256 tokenTwoDecimals = ERC20(uniswapPoolsToSettings[_poolAddress].tokenTwo).decimals();
        uniswapPoolsToSettings[_poolAddress].tokenTwoBaseUnit = 10 ** tokenTwoDecimals;
        uniswapPoolsToSettings[_poolAddress].isValid = true;

        allowedUniswapPools.push(_poolAddress);
    }
```

```
function removePool(address _poolAddress) external onlyOwner {
    require (
        uniswapPoolsToSettings[_poolAddress].isValid,
        "Uniswap pool address does not exist"
    );

    allowedUniswapPools = allowedUniswapPools.remove(_poolAddress);
    delete uniswapPoolsToSettings[_poolAddress];
}

function getAllowedUniswapPools() external view returns (address[] memory) {
    return allowedUniswapPools;
}



/* ============ Internal Functions ============ */

function _getUniswapPrice(
    IPriceOracle _priceOracle,
    address _poolAddress,
    address _masterQuoteAsset
)
    internal
    view
    returns (uint256)
{
    PoolSettings memory poolInfo = uniswapPoolsToSettings[_poolAddress];
    IUniswapV2Pair poolToken = IUniswapV2Pair(_poolAddress);

    // Get prices against master quote asset. Note: if prices do not exist, function will revert
    uint256 tokenOnePriceToMaster = _priceOracle.getPrice(poolInfo.tokenOne, _masterQuoteAsset);
    uint256 tokenTwoPriceToMaster = _priceOracle.getPrice(poolInfo.tokenTwo, _masterQuoteAsset);

    // Get reserve amounts
    (
        uint256 tokenOneReserves,
        uint256 tokenTwoReserves
    ) = UniswapV2Library.getReserves(uniswapFactory, poolInfo.tokenOne, poolInfo.tokenTwo);
```

```
        uint256 normalizedTokenOneBaseUnit = tokenOneReserves.preciseDiv(poolInfo.tokenOneBaseUnit);
        uint256 normalizedTokenBaseTwoUnits = tokenTwoReserves.preciseDiv(poolInfo.tokenTwoBaseUnit);

        uint256 totalNotionalToMaster =
normalizedTokenOneBaseUnit.preciseMul(tokenOnePriceToMaster).add(normalizedTokenBaseTwoUnits.preciseMul(tokenT
woPriceToMaster));
        uint256 totalSupply = poolToken.totalSupply();

        return totalNotionalToMaster.preciseDiv(totalSupply);
    }
}
```

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github
https://github.com/slowmist