KNOWNSEC

Degree of secrecy ★ | Project team open

# Smart Contract Audit Report

Security status

## Safe

★ ★ ★ ★ ★

Principal tester: knownsec blockchain security team

# Release notes

| Revise the | time | Revised by | The version |
|---|---|---|---|
| Written document | **20201015** | **knownsec blockchain security team** | V1.0 |

# Document information

| The document name | Document | The document number | Level of |
|---|---|---|---|
| **BTF Smart contract audit report** | V1.0 | **BTF-ZNNY-20201015** | **Open project Team** |

# The statement

KnownSec only issues this report on the facts that have occurred or exist before the issuance of this report, and shall assume the corresponding responsibility therefor.KnownSec is not in a position to judge the security status of its smart contract and does not assume responsibility for the facts that occur or exist after issuance.The security audit analysis and other contents of this report are based solely on the documents and information provided by the information provider to KnownSec as of the issuance of this report.KnownSec assumes that the information provided was not missing, altered, truncated or suppressed.If the information provided is missing, altered, deleted, concealed or reflected in a way inconsistent with the actual situation, KnownSec shall not be liable for any loss or adverse effect caused thereby.

Knownsec

# Directory



knownsec

Knownsec

Knownsec

# 1. Review

The effective test time of this report is from October 14, 2020 to October 15, 2020. During this period, the security and standardization of the BTF smart contract code will be audited and used as the statistical basis for the report.

In this audit, Knownsec engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (see Chapter 3), and found that the problem of excessive owner privileges can be solved by migrating to a timelock contract during deployment; there is a token issuance problem, but because the problem depends on the requirements of the exchange, the overall assessment is **passed**.

**The smart contract security audit results: PASS**

Since this test is conducted in a non-production environment, all codes are updated, the test process is communicated with the relevant interface personnel, and relevant test operations are carried out under the control of operational risks, so as to avoid production and operation risks and code security risks in the test process.

**The target information of this test:**

| entry | description |
|---|---|
| **Project name** | BTF Finance |
| **Contract Address** | https://github.com/btf-finance/btf-contract |
| **Code type** | Token Code, DeFi Protocol Code, Smart Contract Code |
| **Code language** | solidity |

**Contractual Documents and Hash：**

| The contract documen | MD5 |
|---|---|
| Controller.sol | E8B8423CCC94CB068CD405C05E573697 |
| Converter.sol | FCFD90F035BABD46690AF81B13E2687F |
| Gauge.sol | 23228A3B0ED0B65017B34E59B1D6E1D4 |

Knownsec

| | |
|---|---|
| IBTFReferral.sol | 3518D80DC2C87178E465A9B6CB703B2D |
| IController.sol | 11DC2129CBD48BCE49F4D576ED5AAFD8 |
| IMigratorChef.sol | 45E5B260BFD97CED7872DF8E9B1DFFF6 |
| IStakingRewards.sol | 3D193E086B262608FBFDF004F29857FD |
| IStrategy.sol | 2FB2972AFC648E89FAEC2D24248137EA |
| IStrategyConverter.sol | 21014D4CE13C1E44DE8082553BC44611 |
| ISwerveFi.sol | C5CDCF63C5DCB3FFEF098ED4A7DF52B6 |
| IUniswapV2Factory.sol | B66E188E5C62E7C0364632347109B6C8 |
| IVault.sol | C3A5BFA8623AD868A41089D3FC771801 |
| IYfvRewards.sol | 7472729512D2D23A4A1AD89ACEFED504 |
| OneSplitAudit.sol | A9153F106A980DF38CA67874149AAF52 |
| UniswapRouterV2.sol | 6C600BB869C8F21F0F4184310A17FC7B |
| USDT.sol | 5FCA0C8CF94C05071F5D5A0E10C37F23 |
| StrategySwerve.sol | D8773E77AF699B6FA999305E7286BD12 |
| StrategyUniEthDaiLp.sol | 485A761B65DCEA1EDDE46A63E873BCF0 |
| StrategyUniEthUsdcLp.sol | CDADAB02586FC6B2D2F619C6E6E5F5B4 |
| StrategyUniEthUsdtLp.sol | 0839333AA32712DAB733CA7F4C207D7A |
| StrategyUniEthWbtcLp. | 25B5B775B9DA1BA541F1E6DE11BB56BC |

knownsec

| sol | |
|---|---|
| StrategyYfvDai.sol | C8A7D2FD9683D318C486515D86D96F45 |
| StrategyYfvTusd.sol | 42BFC6DA19E50989013920A776713567 |
| StrategyYfvUsdc.sol | 28844FF81ABED106975CF2819E75F25B |
| StrategyYfvUsdt.sol | AE369C3BB3713BA85F83882B44BE340C |
| Vault.sol | 8821E16F07A00F25F08202D9D301A954 |
| BTFReferral.sol | A6CA80338CA56D46D8D14B8DCF3F62AB |
| BTFToken.sol | F7B002C03C6BB5DA99EA246478699527 |
| MasterChef.sol | 4BF6C672B3AA6CC80D04A7BE54067E47 |
| Migrations.sol | CA8D6CA8A6EDF34F149A5095A8B074C9 |
| MockBtfToken.sol | 19754700A2CAF400BD86B2D3A5C3C756 |
| MockERC20.sol | 5801424F9432ACD5B9CAEC7EA3A15F2E |
| Timelock.sol | 7FBD9498672695C34393331F21A51B94 |

Knownsec

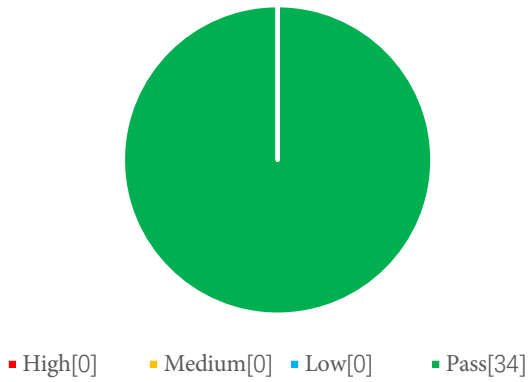
# 2. Code Vulnerability Analysis

## 2.1 Vulnerability level distribution

This vulnerability risk is calculated by level:

| Statistical table of vulnerability risk levels | | | |
|---|---|---|---|
| High Risk | Medium Risk | Low Risk | Pass |
| 0 | 0 | 0 | 34 |

Risk level distribution map




Knownsec

## 2.2 Summary of audit results

| Audit results | | | |
|---|---|---|---|
| Audit Project | Audit content | Status | Description |
| Vault security detection | **Controller contract variables and constructors** | Pass | After testing, there is no such safety problem. |
| | **Controller contract yearn function** | Pass | After testing, there is no such safety problem. |
| | **Vault contract constructor** | Pass | After testing, there is no such safety problem. |
| | **Vault contract deposit function** | Pass | After testing, there is no such safety problem. |
| | **Vault contract withdraw function** | Pass | After testing, there is no such safety problem. |
| | **Strategy contract Harvest function** | Pass | After testing, there is no such safety problem. |
| | **Strategy contract withdraw function** | Pass | After testing, there is no such safety problem. |
| Basic code vulnerability detection | **Compiler version security** | Pass | After testing, there is no such safety problem. |
| | **Redundant code** | Pass | After testing, there is no such safety problem. |
| | **Use of secure arithmetic library** | Pass | After testing, there is no such safety problem. |
| | **Coding Method not recommended** | Pass | After testing, there is no such safety problem. |
| | **Use of require/assert** | Pass | After testing, there is no such safety problem. |
| | **Fallback function security** | Pass | After testing, there is no such safety problem. |
| | **TX. Origin Authentication** | Pass | After testing, there is no such safety problem. |
| | **Owner permission control** | Low risk (Pass) | After testing, the project has deployed with a 12-hour time lock, and therefore consider as low risk overall. |
| | **Gas consumption test** | Pass | After testing, there is no such safety problem. |

| | | | |
|---|---|---|---|
| | call 注入攻击 | Pass | After testing, there is no such safety problem. |
| | **Call injection test** | Pass | After testing, there is no such safety problem. |
| | **Issue of token loopholes** | Low risk<br>(Pass) | The code was tested for the presence of the additional token feature, but as it is subject to exchange requirements, the overall rating was passed. |
| | **Access control detection** | Pass | After testing, there is no such safety problem. |
| | **Numerical overflow detection** | Pass | After testing, there is no such safety problem. |
| | **Arithmetic precision error** | Pass | After testing, there is no such safety problem. |
| | **Error using random number** | Pass | After testing, there is no such safety problem. |
| | **Use of unsafe interfaces** | Pass | After testing, there is no such safety problem. |
| | **Variable coverage** | Pass | After testing, there is no such safety problem. |
| | **Uninitialized storage pointer** | Pass | After testing, there is no such safety problem. |
| | **Return value call validation** | Pass | After testing, there is no such safety problem. |
| | **Transaction order dependence** | Pass | After testing, there is no such safety problem. |
| | **Timestamp dependency attack** | Pass | After testing, there is no such safety problem. |
| | **Denial of service attack** | Pass | After testing, there is no such safety problem. |
| | **Fake recharge vulnerability** | Pass | After testing, there is no such safety problem. |
| | **Reentry attack detection** | Pass | After testing, there is no such safety problem. |
| | **Replay attack detection** | Pass | After testing, there is no such safety problem. |
| | **Rearrangement attack detection** | Pass | After testing, there is no such safety problem. |

# 3. Vault security detection

## 3.1. Controller contract variables and constructors【Pass】

**Audit analysis：** Controller Controller.sol Contractual variable definitions and well-designed constructors

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

    address public constant burn = 0x000000000000000000000000000000000000dEaD;
    address public onesplit = 0xC586BeF4a0992C495Cf22e1aeEE4E446CECDee0E;

    address public governance;//knownsec//  Governance address
    address public strategist;
    address public timelock;

    // community fund
    address public comAddr;//knownsec//  Community address
    // development fund
    address public devAddr;//knownsec//  Dev address
    // burn or repurchase
    address public burnAddr;//knownsec//  Burn address
    mapping(address => address) public vaults;//knownsec//  Address mapping for each vault contract
    mapping(address => address) public strategies;//knownsec//  Address mapping for each token strategy contract
    mapping(address => mapping(address => address)) public converters;//knownsec//  converters
    mapping(address => mapping(address => address)) public strategyConverters;
    mapping(address => mapping(address => bool)) public approvedStrategies;

    uint256 public split = 500;//knownsec//  Vault fee  split/max 5%
uint256 public constant max = 10000;
```

Knownsec

```
    constructor(
        address _governance,//knownsec//  Initialized import governance address
        address _strategist,//knownsec//  Initialized import strategy address
        address _comAddr, // should be the multisig//knownsec//  Initialized import comm address
        address _devAddr,
        address _burnAddr, //should be the multisig
        address _timelock
    ) public {
        governance = _governance;
        strategist = _strategist;
        comAddr = _comAddr;
        devAddr = _devAddr;
        burnAddr = _burnAddr;
        timelock = _timelock;
    }
```

**Safety advice: None.**

## 3.2.Controller contract yearn function【Pass】

**Audit analysis :** Controller Controller.sol contract's yearn is to collect the specified tokens for a given strategy, and earn interest on the remainder after the vault fee is withdrawn from earnings and reinvested in the strategy.

```
function yearn(
        address _strategy,
        address _token,
        uint256 parts
    ) public {
        require(
            msg.sender == strategist || msg.sender == governance,
            "!governance"
        );
        // This contract should never have value in it, but just incase since this is a public call
```

knownsec

```
uint256 _before = IERC20(_token).balanceOf(address(this));
IStrategy(_strategy).withdraw(_token);//knownsec// Withdraw all tokens from the strategy contract to this contract.
uint256 _after = IERC20(_token).balanceOf(address(this));
if (_after > _before) {
    uint256 _amount = _after.sub(_before);//knownsec// Actual value of withdrawals
    address _want = IStrategy(_strategy).want();
    uint256[] memory _distribution;
    uint256 _expected;
    _before = IERC20(_want).balanceOf(address(this));//knownsec// The amount of tokens required for the specified strategy for this contract
    IERC20(_token).safeApprove(onesplit, 0);
    IERC20(_token).safeApprove(onesplit, _amount);//knownsec// Authorize onesplit Differential
    (_expected, _distribution) = OneSplitAudit(onesplit).getExpectedReturn(_token, _want, _amount, parts, 0);
    OneSplitAudit(onesplit).swap(
        _token,
        _want,
        _amount,
        _expected,
        _distribution,
        0
    );//knownsec// _token convert to_want
    _after = IERC20(_want).balanceOf(address(this));//knownsec// Amount of _want tokens for this contract after conversion
    if (_after > _before) {
        _amount = _after.sub(_before);//knownsec// Actual difference before and after conversion
        uint256 _reward = _amount.mul(split).div(max);//knownsec// Reward = Actual difference * split / max
        earn(_want, _amount.sub(_reward));//knownsec// difference - reward For strategic investments
        IERC20(_want).safeTransfer(comAddr, _reward);//knownsec// reward
```

Knownsec

```
transfer to rewards address
                }
            }
        }
```

**Safety advice: None.**

## 3.3.Vault contract constructor【Pass】

**Audit analysis：**Using Vault as an example, the vault class contract provides bToken liquidity mining of other tokens

```
constructor(address _token, address _governance, address _controller)
public
ERC20(
    string(abi.encodePacked("btf ", ERC20(_token).name())),
    string(abi.encodePacked("b", ERC20(_token).symbol()))//knownsec// bToken
)
{
    _setupDecimals(ERC20(_token).decimals());
    token = IERC20(_token);
    governance = _governance;//knownsec// Initial controller contract address
    controller = _controller;
}
```

**Safety advice: None.**

## 3.4.Vualt contract deposit function【Pass】

**Audit analysis：**Using Vault.sol as an example, the Deposit function is used to deposit the specified token and convert it into the corresponding liquid interest-bearing bToken for yield purposes.

```
function deposit(uint256 _amount) public {//knownsec// deposit
        uint256 tokenBalanceWithoutBooster = balance().sub(balanceOf(profitBooster));
```



Knownsec

```
        uint256 totalSupplyWithoutBooster = totalSupply().sub(balanceOf(profitBooster));

        uint256 _before = token.balanceOf(address(this));
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 _after = token.balanceOf(address(this));//knownsec//  Change in calculated balance
        _amount = _after.sub(_before);
        // Additional check for deflationary tokens
        uint256 shares = 0;
        if (totalSupplyWithoutBooster == 0 || msg.sender == profitBooster) {
            shares = _amount;
        } else {
            shares = _amount.mul(totalSupplyWithoutBooster).div(tokenBalanceWithoutBooster);//knownsec// Calculating the liquidity ratio
        }
        _mint(msg.sender, shares);//knownsec//  Casting credentials for callers
    }
```

**Safety advice: None.**

## 3.5. Vualt contract withdraw function【Pass】

**Audit analysis :** Using Vault.sol as an example, the withdraw function is used to withdraw liquidity yield from a strategy contract.

```
    // No rebalance implementation for lower fees and faster swaps
    function withdraw(uint256 _shares) public {
        uint256 tokenBalanceWithoutBooster = balance().sub(balanceOf(profitBooster));
        uint256 totalSupplyWithoutBooster = totalSupply().sub(balanceOf(profitBooster));

        uint256 r = 0;
        if (profitBooster == msg.sender) {
```

knownsec

```
            r = _shares;
        } else {
            r = tokenBalanceWithoutBooster.mul(_shares).div(totalSupplyWithoutBooster);
        }
        _burn(msg.sender, _shares);//knownsec//  Token burning to reduce supply

        // Check balance
        uint256 b = token.balanceOf(address(this));//knownsec//  Balance check
        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);//knownsec//  Token sent to liquidity providers
    }
```

**Safety advice: None.**

## 3.6.Strategy contract harvest function【Pass】

**Audit analysis：**Using StrategySwerve.sol as an example, the harvest function is used to invest interest according to the specified strategy.

```
function harvest() public {//knownsec//Earning liquidity interest
        // Anyone can harvest it at any given time.
        // I understand the possibility of being frontrun
        // But ETH is a dark forest, and I wanna see how this plays out
        // i.e. will be be heavily frontrunned?
```

Knownsec

```
        //          if so, a new strategy will be deployed.

        // stablecoin we want to convert to
        (address to, uint256 toIndex) = getMostPremiumStablecoin();//knownsec//Get as many
stable coins as possible

        // Collects SWRV tokens
        Mintr(mintr).mint(rewards);
        uint256 _swrv = IERC20(swrv).balanceOf(address(this));//knownsec//Calculation swrv
balance of this contract
        if (_swrv > 0) {
            // 10% is locked up for future gov
            if (keepSWRV > 0) {//knownsec//  Deduction for governance costs
                uint256 _keepSWRV = _swrv.mul(keepSWRV).div(keepSWRVMax);
                IERC20(swrv).safeTransfer(
                        IController(controller).devAddr(),
                    _keepSWRV
                );//knownsec//  governance costs transfer to controller
                _swrv = _swrv.sub(_keepSWRV);
            }

            _swap(swrv, to, _swrv);
        }

        uint256 _to = IERC20(to).balanceOf(address(this));
        if (_to > 0) {
            // Burn some btfs first
            if (burnFee > 0) {//knownsec// Burn fee
                uint256 _burnFee = _to.mul(burnFee).div(burnMax);
                _swap(to, btf, _burnFee);
                IERC20(btf).transfer(
                    IController(controller).burnAddr(),
                    IERC20(btf).balanceOf(address(this))
```



Knownsec

```
                );
                _to = _to.sub(_burnFee);
            }
        }

        // Adds in liquidity for swusdv2 pool to get back want (swusd)
        if (_to > 0) {
            IERC20(to).safeApprove(curve, 0);
            IERC20(to).safeApprove(curve, _to);
            uint256[4] memory liquidity;
            liquidity[toIndex] = _to;
            ISwerveFi(curve).add_liquidity(liquidity, 0);
        }

        // We want to get back swrv
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            // 4.5% rewards gets sent to treasury
            IERC20(want).safeTransfer(
                IController(controller).comAddr(),
                _want.mul(performanceFee).div(performanceMax)
            );

            deposit();// knownsec //Transfer of yield
        }
    }
```

**Safety advice: None.**

## 3.7.Strategy contract withdraw function【Pass】

**Audit analysis：** Using StrategySwerve.sol as an example, withdraw functions are used to provide liquidity tokens to the vault contract.

```
// Withdraw partial funds, normally used with a vault withdrawal
    function withdraw(uint256 _amount) external {//knownsec// withdraw assets _amount
        require(msg.sender == controller, "!controller");//knownsec// Controller calls only
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {//knownsec// Processing insufficient balances
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }

        if (withdrawalFee > 0) {//knownsec// Processing fees for withdrawals
            uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
            IERC20(want).safeTransfer(IController(controller).comAddr(), _fee);
            _amount = _amount.sub(_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);//knownsec// Transfer
    }

    // Withdraw all funds, normally used when migrating strategies
    function withdrawAll() external returns (uint256 balance) {//knownsec// Withdrawal
of all assets, controller use
        require(msg.sender == controller, "!controller");
        _withdrawAll();

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

        address _vault = IController(controller).vaults(address(want));
```

knownsec

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

    function _withdrawAll() internal {
        _withdrawSome(balanceOfPool());
    }

    function _withdrawSome(uint256 _amount) internal returns (uint256) {//knownsec//
Partial withdrawal   Internal use
        Gauge(rewards).withdraw(_amount);
        return _amount;
    }
```

**Safety advice**: None.

knownsec

# 4. Basic code vulnerability detection

## 4.1. Compiler version security【Pass】

Check that a secure compiler version is used in the contract code implementation

**Detection results: After detection, the compiler version of the smart contract code is more than 0.5.8, there is no such security problem.**

**Safety advice: None.**

## 4.2. Redundant code【Pass】

Check if the contract code implementation contains redundant code

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice**: None.

## 4.3. Use of secure arithmetic library【Pass】

Check if the SafeMath security arithmetic library is used in the contract code implementation

**Detection results: The SafeMath security arithmetic library has been used in the smart contract code. There is no security problem.**

**Safety advice: None.**

## 4.4. Coding Method not recommended【Pass】

Check the contract code implementation to see if there are any officially recommended or deprecated encoding options

**Detection results: After detection, there is no security problem in the smart contract code.**

knownsec

**Safety advice: None.**

## 4.5.Use of require/Assert【Pass】

Check the reasonableness of the use of require and Assert statements in your contractual code implementation

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.6.fallback function security【Pass】

Check that the Fallback function is used correctly in the contract code implementation

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.7.TX. Origin Authentication【Pass】

Tx. origin, a global variable that iterates over the entire call stack and returns the address of the account that originally sent the call (or transaction).Using this variable for authentication in a smart contract makes the contract vulnerable to phishing attacks.

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.8.Owner permission control【Low risk】

Check if the Owner in the contract code implementation has too many permissions.For example, arbitrarily modify other account balances, etc.

Knownsec

**Detection results: After detection, the project has deployed with a 12-hour time lock, and therefore consider as low risk overall.**

**Safety advice: None.**

## 4.9.Gas consumption test【Pass】

Check whether the gas consumption exceeds the block maximum limit

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.10. Call injection attack【Pass】

When the call function is called, strict permission control should be done, or the dead call function should be written directly.

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

Knownsec

## 4.11. Low-level function security【Pass】

Check for security vulnerabilities caused by the use of the low-level functions called/Delegatecall used by the contract code implementation

The execution context of the call function is in the contract being invoked; The execution context of the Delegatecall function is in the contract where the function is currently called

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.12. Issue of token loopholes【Low risk】

Check whether there is a function that may increase the total amount of tokens in the token contract after initializing the total amount of tokens.

**Detection result: After detection, the problem exists in the smart contract code, in line 15 of BTFToken.sol**

```
    function mint(address _to, uint256 _amount) public onlyOwner {//    knownsec//
Issuance of additional tokens

        _mint(_to, _amount);
        _moveDelegates(address(0), _delegates[_to], _amount);
    }
```

**Safety advice: the issue is not a security issue, but some exchanges will restrict the use of the incremental function, depending on the requirements of the exchange.**

knownsec

## 4.13. Access control detection【Pass】

Reasonable permissions should be set for different functions in the contract

Check whether the functions in the contract have correctly used keywords such as public and private for visibility modification, and check whether the contract has correctly defined and used modifier to restrict access to key functions, so as to avoid problems caused by overstepping authority.

**Detection result: After detection, there is no security problem in the smart contract code.**

**Security advice: None.**

## 4.14. Numerical overflow detection【Pass】

The arithmetic problem in smart contract refers to integer overflow and integer underflow.

Instead of trying to contain something deep inside the body, which is capable of processing a maximum of 256 digits (2^256-1), a maximum increase of 1 would allow the body to drain down to zero. Similarly, when the number is unsigned, 0 minus 1 overflows to get the maximum number value.

Integer overflow and underflow are not a new type of vulnerability, but they are particularly dangerous in smart contracts. Overflow scenarios can lead to incorrect results, especially if the possibility is not anticipated, and can affect the reliability and security of the program.

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.15. Arithmetic precision error【Pass】

Solidity as a programming language and common programming language similar data structure design, such as: variables, constants, and functions, arrays, functions, structure and so on, Solidity and common programming language also has a



Knownsec

larger difference - no floating-point Solidity, Solidity and all the numerical computing results can only be an integer, decimal will not happen, also not allowed to define the decimal data type. The numerical calculation in the contract is essential, and the design of numerical calculation may cause relative error, such as the same-level calculation: 5/2*10=20, and 5*10/2=25, resulting in error, and the error will be larger and more obvious when the data is larger.

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.16. Error using random number【Pass】

In smart contracts may need to use a random number, although the Solidity of functions and variables can access the value of the unpredictable obviously such as block. The number and block. The timestamp, but they usually or more open than it looks, or is affected by the miners, that is, to some extent, these random Numbers is predictable, so a malicious user can copy it and usually rely on its unpredictability to attack the function.

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.17. Use of unsafe interfaces【Pass】

Check whether unsafe interfaces are used in the contract code implementation

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.18. Variable coverage【Pass】

Check the contract code implementation for security issues caused by variable overwriting

**Detection results: After detection, there is no security problem in the smart contract code.**

Knownsec

**Safety advice: None.**

## 4.19. Uninitialized storage pointer【Pass】

A special data structure is allowed to be struct in Soldesert, and local variables inside the function are stored in storage or memory by default.

Presence of storage and memory are two different concepts, which would involve trying to involve a pointer to an uninitialized reference, whereas an uninitialized local storage would cause variables to point to other stored variables, leading to variable overwrite, or even more serious consequences, and struct variables should be avoided in development from initializing struct variables in functions.

**Detection results: After detection, the smart contract code does not use the structure, there is no such problem.**

**Safety advice: None.**

## 4.20. Return value call validation【Pass】

This problem occurs mostly in smart contracts associated with currency transfers, so it is also known as silent failed or unchecked send.

A transfer method, such as transfer(), send(), or call.value(), could all be used to send Ether to an address, with the difference between throw and state rollback if the transfer fails; Only 2300GAS will be Pass for invocation to prevent reentrant attack; Send returns false on failure;Only 2300GAS will be Pass for invocation to prevent reentrant attack;Call.value returns false on failure;Passing all available gas for invocation (which can be restricted by Passing in the GAS_value parameter) does not effectively prevent a reentrant attack.

If the return value of the send and call.value transfer function above is not checked in the code, the contract will continue to execute the following code, possibly causing unexpected results due to the failure of Ether to send.

Knownsec

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.21. Transaction order dependence【Pass】

Since miners always get gas fees through a code that represents an externally owned address (EOA), users can specify higher fees for faster transactions. Because the Ethereum blockchain is public, everyone can see the content of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transaction at a higher cost to preempt the original solution.

**Detection result: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.22. Timestamp dependency attack【Pass】

The timestamp of the data block usually USES the miner's local time, which can fluctuate in the range of about 900 seconds. When other nodes accept a new block, they only need to verify that the timestamp is later than the previous block and within 900 seconds of the local time. A miner can profit by setting the timestamp of the block to meet conditions as favorable to him as possible.

Check to see if there is any key functionality that depends on the timestamp in the contract code implementation

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

Knownsec

## 4.23. Denial of service attack【Pass】

In Ethereum's world, denial of service is deadly, and smart contracts that suffer from this type of attack may never return to normal functioning. The reasons for smart contract denial of service can be many, including malicious behavior while on the receiving end of a transaction, gas depletion due to the artificial addition of needed gas for computing functions, abuse of access control to access private components of smart contracts, exploitation of obtuse and negligence, and so on.

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.24. Fake recharge vulnerability【Pass】

In the transfer function of the token contract, the balance check over the originator (MSG. sender) becomes an if judgment method. When the veto [MSg. sender] < value, enter the else logic part and return false, no exception will become available. We believe that the if/else gentle judgment method is an unrigorous coding method in the transfer sensitive function scene.

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.25. Reentry attack detection【Pass】

Re-entry holes are The most famous ethereum smart contract holes that have led to The DAO hack of Ethereum.

The call.value() function in Soldesert consumes all the gas it receives when it is used to send Ether, and there is a risk of a reentrant attack if the operation to send Ether is called to the call.value() function before it actually reduces the balance in the sender's account.

Knownsec

**Detection results: After detection, there is no security problem in the smart contract code.**

**Safety advice: None.**

## 4.26. Replay attack detection【Pass】

If the requirement of delegation management is involved in the contract, attention should be paid to the non-reusability of verification to avoid replay attack

In the asset management system, there are often cases of entrusted management in which the principal gives the assets to the agent for management and the principal pays a certain fee to the agent. This business scenario is also common in smart contracts.

**Detection results: After detection, the call function is not used in the smart contract, so there is no such vulnerability.**

**Safety advice: None.**

## 4.27. Rearrangement attack detection【Pass】

A rearrangement attack is an attempt by a miner or other party to "compete" with an smart contract participant by inserting their information into a list or mapping, thereby giving the attacker an opportunity to store their information in the contract.

**Detection results: After detection, there is no relevant vulnerability in the smart contract code.**

**Safety advice: None.**

Knownsec

# 5. Appendix A: Contract code

Source code for this test：

***Controller.sol***

```
pragma solidity ^0.6.7;//knownsec//  Specifying the compiler version

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

import "../interface/IStrategy.sol";
import "../interface/IVault.sol";
import "../interface/Converter.sol";
import "../interface/IStrategyConverter.sol";
import "../interface/OneSplitAudit.sol";


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

    address public constant burn =
0x000000000000000000000000000000000000dEaD;
address public onesplit = 0xC586BeF4a0992C495Cf22e1aeEE4E446CECDee0E;

address public governance;//knownsec//  Governance address
address public strategist;
address public timelock;

// community fund
address public comAddr;//knownsec//  Community address
// development fund
address public devAddr;//knownsec//  Dev address
// burn or repurchase
address public burnAddr;//knownsec//  Burn token address
mapping(address => address) public vaults;//knownsec//  Address mapping for each vault contract
mapping(address => address) public strategies;//knownsec//  Address mapping for each token strategy contract
mapping(address => mapping(address => address)) public converters;//knownsec//  converters
mapping(address => mapping(address => address)) public strategyConverters;
mapping(address => mapping(address => bool)) public approvedStrategies;

    uint256 public split = 500;//knownsec//  vault fee    split/max 5%
uint256 public constant max = 10000;

    constructor(
        address _governance,//knownsec//  Initialized import governance address
        address _strategist,//knownsec//  Initialized import strategy address
        address _comAddr, // should be the multisig      //knownsec//  Initialized import comm address
        address _devAddr,
        address _burnAddr, //should be the multisig
        address _timelock
    ) public {
        governance = _governance;
        strategist = _strategist;
        comAddr = _comAddr;
        devAddr = _devAddr;
        burnAddr = _burnAddr;
        timelock = _timelock;
    }

    function setComAddr(address _comAddr) public {//knownsec//  Change community address, governed address call only
        require(msg.sender == governance, "!governance");
        comAddr = _comAddr;
    }

    function setDevAddr(address _devAddr) public {//knownsec//  Change developer address, governed address calls only
        require(msg.sender == governance || msg.sender == devAddr, "!governance");
        devAddr = _devAddr;
    }

    function setBurnAddr(address _burnAddr) public {//knownsec//  Change burn address, governed address call only
        require(msg.sender == governance, "!governance");
        burnAddr = _burnAddr;
    }
```

knownsec

```
    function setStrategist(address _strategist) public {//knownsec// Change strategy address, governed address call only
        require(msg.sender == governance, "!governance");
        strategist = _strategist;
    }

    function setSplit(uint256 _split) public {
        require(msg.sender == governance, "!governance");//knownsec// Change the split ratio address, govern address calls only
        split = _split;
    }

    function setOneSplit(address _onesplit) public {//knownsec// Change the split ratio address, govern address calls only
        require(msg.sender == governance, "!governance");
        onesplit = _onesplit;
    }

    function setGovernance(address _governance) public {//knownsec// Change govern address, govern address calls only
        require(msg.sender == governance, "!governance");
        governance = _governance;
    }

    function setTimelock(address _timelock) public {//knownsec// Set the timelock management address, only timelock management address calls.
        require(msg.sender == timelock, "!timelock");
        timelock = _timelock;
    }

    function setVault(address _token, address _vault) public {//knownsec// Adding new tokens and policies, only governance and policy management address calls
        require(
            msg.sender == strategist || msg.sender == governance,
            "!strategist"
        );
        require(vaults[_token] == address(0), "vault");
        vaults[_token] = _vault;
    }

function approveStrategy(address _token, address _strategy) public {//knownsec// Enable strategy, timelock management address calls only

        require(msg.sender == timelock, "!timelock");
        approvedStrategies[_token][_strategy] = true;
    }

function revokeStrategy(address _token, address _strategy) public {//knownsec// Stop strategy, timelock management address calls only

        require(msg.sender == governance, "!governance");
        approvedStrategies[_token][_strategy] = false;
    }

    function setConverter(//knownsec// Set the converter address, governance address and strategy management address calls only
        address _input,
        address _output,
        address _converter
    ) public {
        require(
            msg.sender == strategist || msg.sender == governance,
            "!strategist"
        );
        converters[_input][_output] = _converter;
    }

    function setStrategyConverter(
        address[] memory stratFrom,
        address[] memory stratTo,
        address _stratConverter
) public {//knownsec// Set strategy switch address, governance address only and strategy management address call.
        require(
            msg.sender == strategist || msg.sender == governance,
            "!strategist"
        );

        for (uint256 i = 0; i < stratFrom.length; i++) {
            for (uint256 j = 0; j < stratTo.length; j++) {
                strategyConverters[stratFrom[i]][stratTo[j]] = _stratConverter;
            }
        }
    }

    function setStrategy(address _token, address _strategy) public {//knownsec// Adding new strategy addresses, governance only and strategy management address calls
        require(
            msg.sender == strategist || msg.sender == governance,
            "!strategist"
        );
        require(approvedStrategies[_token][_strategy] == true, "!approved");

        address _current = strategies[_token];
```

knownsec

```
        if (_current != address(0)) {//knownsec//If the previous strategy exists, then withdraw all funds first.
            IStrategy(_current).withdrawAll();
        }
        strategies[_token] = _strategy;
    }

    function earn(address _token, uint256 _amount) public {//knownsec// Deposti
        address _strategy = strategies[_token];//knownsec //Get the contract address of the strategy
        address _want = IStrategy(_strategy).want();//knownsec //The token address required by the strategy
        if (_want != _token) {//knownsec //If the strategy needs to be different from the input, it needs to be converted first.
            address converter = converters[_token][_want];//knownsec //Converter contract address
            IERC20(_token).safeTransfer(converter, _amount);//knownsec //Transfer to converters
            _amount = Converter(converter).convert(_strategy);//knownsec //Perform conversions
            IERC20(_want).safeTransfer(_strategy, _amount);
        } else {
            IERC20(_token).safeTransfer(_strategy, _amount);
        }
        IStrategy(_strategy).deposit();//knownsec //Deposit
    }

    function balanceOf(address _token) external view returns (uint256) {//knownsec// Get the balance of the token in the strategy
        return IStrategy(strategies[_token]).balanceOf();
    }

    function withdrawAll(address _token) public {//knownsec// Withdrawal of all balances for a given token, with only governance and
strategy management address calls
        require(
            msg.sender == strategist || msg.sender == governance,
            "!strategist"
        );
        IStrategy(strategies[_token]).withdrawAll();
    }

function inCaseTokensGetStuck(address _token, uint256 _amount) public {//knownsec //Convert any erc20
        require(
            msg.sender == strategist || msg.sender == governance,
            "!governance"
        );
        IERC20(_token).safeTransfer(msg.sender, _amount);//knownsec// Transfer of designated tokens to a governance address
    }   account

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

    function getExpectedReturn(
        address _strategy,
        address _token,
        uint256 parts
    ) public view returns (uint256 expected) {
        uint256 _balance = IERC20(_token).balanceOf(_strategy);//knownec //Get the balance of a token from a strategist
        address _want = IStrategy(_strategy).want();//knownec //Get the address of the token needed by the strategist
        (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 == strategist || 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));
        IStrategy(_strategy).withdraw(_token);//knownsec// All tokens from a strategy contract are withdrawn to this contract.
        uint256 _after = IERC20(_token).balanceOf(address(this));
        if (_after > _before) {
            uint256 _amount = _after.sub(_before);//knownsec// Actual value of withdrawals
            address _want = IStrategy(_strategy).want();
            uint256[] memory _distribution;
            uint256 _expected;
            _before = IERC20(_want).balanceOf(address(this));//knownsec// The amount of tokens required for the specified
            strategy for this contract
```

Knownsec

```
                IERC20(_token).safeApprove(onesplit, 0);
                IERC20(_token).safeApprove(onesplit, _amount);//knownsec// Authorize onesplit Differential
                (_expected, _distribution) = OneSplitAudit(onesplit).getExpectedReturn(_token, _want, _amount,
parts, 0);
                OneSplitAudit(onesplit).swap(
                    _token,
                    _want,
                    _amount,
                    _expected,
                    _distribution,
                    0
                );//knownsec// token convert to _want
                _after = IERC20(_want).balanceOf(address(this));//knownsec// Amount of _want tokens for this contract after conversion
                if (_after > _before) {
                    _amount = _after.sub(_before);//knownsec// Actual difference before and after conversion
                    uint256 _reward = _amount.mul(split).div(max);//knownsec// Reward = Actual difference * split / max
                    earn(_want, _amount.sub(_reward));//knownsec// Difference - Reward For strategic investments
                    IERC20(_want).safeTransfer(comAddr, _reward);//knownsec// Reward transfer to rewards address
                }
            }
        }

        function withdraw(address _token, uint256 _amount) public {//knownsec// Withdraw a certain amount of the specified tokens,
only the token bank address call.
            require(msg.sender == vaults[_token], "!vault");
            IStrategy(strategies[_token]).withdraw(_amount);
        }

        // Swaps between vaults
        // Note: This is supposed to be called
        //       by a user if they'd like to swap between vaults w/o the 0.5% fee
        function userSwapVault(
            address _fromToken,
            address _toToken,
            uint256 _pAmount // Pickling token amount to convert
        ) public nonReentrant returns (uint256) {//knownsec// Users make token swaps
            address _fromVault = vaults[_fromToken];
            address _toVault = vaults[_toToken];

            address _fromStrategy = strategies[_fromToken];
            address _toStrategy = strategies[_toToken];


            address _strategyConverter = strategyConverters[_fromStrategy][_toStrategy];

            require(_strategyConverter != address(0), "!strategy-converter");

            // 1. Transfers bVault tokens from msg.sender
            IVault(_fromVault).transferFrom(msg.sender, address(this), _pAmount);//knownsec// Receive tokens from swappers

            // 2. Get amount of tokens to transfer from strategy to burn
            // Note: this token amount is the LP token
            uint256 _fromTokenAmount = IVault(_fromVault).getRatio().mul(_pAmount).div(
                1e18
            );//knownsec// Calculating the LP_token Scale

            // If we don't have enough funds in the strategy
            // We'll deposit funds from the vault to the strategy
            // Note: This assumes that no single person is responsible
            //       for 100% of the liquidity.
            // If this a single person is 100% responsible for the liquidity
            // we can simply set min = max in vaults
            if (IStrategy(_fromStrategy).balanceOf() < _fromTokenAmount) {//knownsec// Processing vault token insufficiencies
                IVault(_fromVault).earn();
            }

            // 3. Withdraw tokens from strategy and burns pToken
            IVault(_fromVault).transfer(burn, _pAmount);//knownsec// Burn token from the vault.
            IStrategy(_fromStrategy).freeWithdraw(_fromTokenAmount);

            // 4. Converts to Token
            IERC20(_fromToken).approve(_strategyConverter, _fromTokenAmount);
            IStrategyConverter(_strategyConverter).convert(
                msg.sender,
                _fromToken,
                _toToken,
                _fromTokenAmount
            );//knownsec// convert token

            // 5. Deposits into BFTVault
            uint256 _toTokenAmount = IERC20(_toToken).balanceOf(address(this));//knownsec// Get the address token balance

            IERC20(_toToken).approve(_toVault, _toTokenAmount);
            IVault(_toVault).deposit(_toTokenAmount);//knownsec// deposit

            // 6. Sends msg.sender all the btf vault tokens
            uint256 _retPAmount = IVault(_toVault).balanceOf(address(this));
```

```
            IVault(_toVault).transfer(
                msg.sender,
              _retPAmount
            );//knownsec// Transfers to depository receipts

            return _retPAmount;
        }
}
```

***Vault.sol***

```
pragma solidity ^0.6.7;

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

import "../interface/IController.sol";

contract Vault is ERC20 {
    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;
    // todo
    address public constant profitBooster = address(0);

    constructor(address _token, address _governance, address _controller)
    public
    ERC20(
        string(abi.encodePacked("btf ", ERC20(_token).name())),
        string(abi.encodePacked("b", ERC20(_token).symbol()))
    )
    {
        _setupDecimals(ERC20(_token).decimals());
        token = IERC20(_token);
        governance = _governance;
        controller = _controller;
    }

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

    function balanceOfToken() public view returns (uint256) {
        if (profitBooster == msg.sender) {
            return balanceOf(profitBooster);
        } else {
            return (balance().sub(balanceOf(profitBooster)))
            .mul(balanceOf(msg.sender))
            .div(totalSupply().sub(balanceOf(profitBooster)));
        }
    }

    function setMin(uint256 _min) external {
        require(msg.sender == governance, "!governance");
        min = _min;
    }

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

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

    // Custom logic in here for how much the token 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 {
```

knownsec

```
        uint256 _bal = available();
        token.safeTransfer(controller, _bal);
        IController(controller).earn(address(token), _bal);
    }

    function depositAll() external {//knownsec// Deposit all tokens of the caller
        deposit(token.balanceOf(msg.sender));
    }

    function deposit(uint256 _amount) public {//knownsec// Deposit
        uint256 tokenBalanceWithoutBooster = balance().sub(balanceOf(profitBooster)); uint256
        totalSupplyWithoutBooster = totalSupply().sub(balanceOf(profitBooster));

        uint256 _before = token.balanceOf(address(this));
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 _after = token.balanceOf(address(this));//knownsec// Calculation of changes in balance
        _amount = _after.sub(_before);
        // Additional check for deflationary tokens
        uint256 shares = 0;
        if (totalSupplyWithoutBooster == 0 || msg.sender == profitBooster) {
            shares = _amount;
        } else {
            shares = _amount.mul(totalSupplyWithoutBooster).div(tokenBalanceWithoutBooster);//knownsec//
Calculation of pool ratio
        }
        _mint(msg.sender, shares);//knownsec// Casting credentials for callers
    }

    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 {
        uint256 tokenBalanceWithoutBooster = balance().sub(balanceOf(profitBooster));
        uint256 totalSupplyWithoutBooster = totalSupply().sub(balanceOf(profitBooster));

        uint256 r = 0;
        if (profitBooster == msg.sender) {
            r = _shares;
        } else {
            r = tokenBalanceWithoutBooster.mul(_shares).div(totalSupplyWithoutBooster);
        }
        _burn(msg.sender, _shares);//knownsec// Burning to reduce supply

        // Check balance
        uint256 b = token.balanceOf(address(this));//knownsec// check balance
        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);//knownsec// Assets sent to liquidity providers
    }

    function getRatio() public view returns (uint256) {
        return balance().mul(1e18).div(totalSupply());
    }
}
```

***StrategySwerve.sol***

```
pragma solidity ^0.6.2;

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

import "../interface/IController.sol";
import "../interface/IStrategy.sol";
import "../interface/Gauge.sol";
import "../interface/ISwerveFi.sol";
import "../interface/UniswapRouterV2.sol";
```

Knownsec

```
contract StrategySwerve {
    using SafeERC20 for IERC20;
using Address for address;
using SafeMath for uint256;

    // gauge of swerve
address public constant rewards = 0xb4d0C929cD3A1FbDc6d57E7D3315cF0C4d6B4bFa;

    // swUSD lp tokens
address public constant want = 0x77C6E4a580c0dCE4E5c7a17d0bc077188a83A059;

    // tokens we're farming
address public constant swrv = 0xB8BAa0e4287890a5F79863aB62b7F175ceCbD433;

    // swusdv2 pool
address public constant curve = 0x329239599afB305DA0A2eC69c58F8a6697F9F88d;

    // swerve minter
address constant public mintr = 0x2c988c3974AD7E604E276AE0294a7228DEf67974;//knownsec// miners

    // stablecoins
address public constant dai = 0x6B175474E89094C44Da98b954EedeAC495271d0F;
address public constant usdc = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
address public constant usdt = 0xdAC17F958D2ee523a2206206994597C13D831ec7;
address public constant tusd = 0x0000000000085d4780B73119b644AE5ecd22b376;

    // weth
address public constant weth = 0xc778417E063141139Fce010982780140Aa0cD5Ab;

    // dex
address public constant univ2Router2 = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;

    // Fees 5% in total
// - 1.5%    keepSWRV for development fund
// - 2%      performanceFee for community fund
// - 1.5%    used to burn/repurchase btfs
uint256 public keepSWRV = 150;
uint256 public constant keepSWRVMax = 10000;

    uint256 public performanceFee = 200;
uint256 public constant performanceMax = 10000;

    uint256 public burnFee = 150;
uint256 public constant burnMax = 10000;

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

address public governance;
address public controller;
address public strategist;
address public timelock;
address public btf;

    constructor(
        address _governance,
        address _strategist,
        address _controller,
        address _timelock,
        address _btf
    ) public {
        governance = _governance;
        strategist = _strategist;
        controller = _controller;
        timelock = _timelock;
        btf = _btf;
    }//knownsec// initialization contract，set governance、strategist、controller、timelock、btf address

    // **** Views ****

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

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

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

    function getName() external pure returns (string memory) {
        return "StrategySwerve";
    }

    function getHarvestable() external returns (uint256) {
```

knownsec

```
        return Gauge(rewards).claimable_tokens(address(this));
    }

    // **** Setters ****

    function setKeepSWRV(uint256 _keepSWRV) external {
        require(msg.sender == governance, "!governance");
        keepSWRV = _keepSWRV;
    }

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

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

    function setBurnFee(uint256 _burnFee) external {
        require(msg.sender == governance, "!governance");
        burnFee = _burnFee;
    }

    function setStrategist(address _strategist) external {
        require(msg.sender == governance, "!governance");
        strategist = _strategist;
    }

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

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

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

    function getMostPremiumStablecoin() public view returns (address, uint256) {// knownsec  Get the most stable coins

        uint256[] memory balances = new uint256[](4);
        // DAI
        balances[0] = ISwerveFi(curve).balances(0);
        // USDC
        balances[1] = ISwerveFi(curve).balances(1).mul(10 ** 12);
        // USDT
        balances[2] = ISwerveFi(curve).balances(2).mul(10 ** 12);
        // TUSD
        balances[3] = ISwerveFi(curve).balances(3);

        // DAI
        if (
            balances[0] < balances[1] &&
            balances[0] < balances[2] &&
            balances[0] < balances[3]
        ) {
            return (dai, 0);
        }

        // USDC
        if (
            balances[1] < balances[0] &&
            balances[1] < balances[2] &&
            balances[1] < balances[3]
        ) {
            return (usdc, 1);
        }

        // USDT
        if (
            balances[2] < balances[0] &&
            balances[2] < balances[1] &&
            balances[2] < balances[3]
        ) {
            return (usdt, 2);
        }

        // TUSD
        if (
            balances[3] < balances[0] &&
```

knownsec

```
                balances[3] < balances[1] &&
                balances[3] < balances[2]
            ) {
                return (tusd, 3);
            }

            // If they're somehow equal, we just want DAI return (dai, 0);

        }

    // **** State Mutations ****

    function deposit() public {//knownsec// Liquidity Mining Token Transfer
            uint256 _want = IERC20(want).balanceOf(address(this));//knownsec Calculate the output token balance of this contract
            if (_want > 0) {
                IERC20(want).safeApprove(rewards, 0);// knownsec Addressing transactional dependencies set 0
                IERC20(want).approve(rewards, _want);
                Gauge(rewards).deposit(_want);
            }
        }

        // 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(swrv != address(_asset), "swrv");

            balance = _asset.balanceOf(address(this));
            _asset.safeTransfer(controller, balance);
        }

        // Contoller only function for withdrawing for free
        // This is used to swap between vaults
        function freeWithdraw(uint256 _amount) external {// knownsec Inter-vault token swap
            require(msg.sender == controller, "!controller");
            uint256 _balance = IERC20(want).balanceOf(address(this));
            if (_balance < _amount) {
                _amount = _withdrawSome(_amount.sub(_balance));
                _amount = _amount.add(_balance);
            }
            IERC20(want).safeTransfer(msg.sender, _amount);
        }

        // Withdraw partial funds, normally used with a vault withdrawal
        function withdraw(uint256 _amount) external {//knownsec// To withdraw assets to vault volume _amount controller uses
            require(msg.sender == controller, "!controller");//knownsec// Controller calls only
            uint256 _balance = IERC20(want).balanceOf(address(this));
            if (_balance < _amount) {//knownsec//Processing insufficient balances
                _amount = _withdrawSome(_amount.sub(_balance));
                _amount = _amount.add(_balance);
            }

            if (withdrawalFee > 0) {//knownsec// Processing fees for withdrawals
                uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
                IERC20(want).safeTransfer(IController(controller).comAddr(), _fee);
                _amount = _amount.sub(_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);//knownsec// Transfer
        }

        // Withdraw all funds, normally used when migrating strategies
    function withdrawAll() external returns (uint256 balance) {//knownsec// To withdraw all assets, the controller uses
            require(msg.sender == controller, "!controller");
            _withdrawAll();

            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 _withdrawAll() internal {
            _withdrawSome(balanceOfPool());
        }

        function _withdrawSome(uint256 _amount) internal returns (uint256) {//knownsec// Partial withdrawals, Internal use
            Gauge(rewards).withdraw(_amount);
            return _amount;
        }
```

Knownsec

```
function brine() public {
        harvest();
    }

    function harvest() public {//knownsec//Earning liquidity interest
        // Anyone can harvest it at any given time.
        // I understand the possibility of being frontrun
        // But ETH is a dark forest, and I wanna see how this plays out
        // i.e. will be be heavily frontrunned?
           //          if so, a new strategy will be deployed.

        // stablecoin we want to convert to
        (address to, uint256 toIndex) = getMostPremiumStablecoin();//knownsec//Get as many stable coins as possible

        // Collects SWRV tokens
        Mintr(mintr).mint(rewards);
              uint256 _swrv = IERC20(swrv).balanceOf(address(this));//knownsec//Calculate the swrv balance of this contract
        if (_swrv > 0) {
            // 10% is locked up for future gov
            if (keepSWRV > 0) {//knownsec// Deduction for governance costs
                uint256 _keepSWRV = _swrv.mul(keepSWRV).div(keepSWRVMax);
                IERC20(swrv).safeTransfer(
                IController(controller).devAddr(),
                      _keepSWRV
                );//knownsec// Governance costs are transferred to the controller
                _swrv = _swrv.sub(_keepSWRV);
            }

            _swap(swrv, to, _swrv);
        }

        uint256 _to = IERC20(to).balanceOf(address(this));
        if (_to > 0) {
            // Burn some btfs first
            if (burnFee > 0) {//knownsec// Burn fee
                uint256 _burnFee = _to.mul(burnFee).div(burnMax);
                  swap(to, btf, _burnFee);
                IERC20(btf).transfer(
                      IController(controller).burnAddr(),
                      IERC20(btf).balanceOf(address(this))
                );
                _to = _to.sub(_burnFee);
            }
        }

        // Adds in liquidity for swusdv2 pool to get back want (swusd)
        if (_to > 0) {
            IERC20(to).safeApprove(curve, 0);
            IERC20(to).safeApprove(curve, _to);
            uint256[4] memory liquidity;
            liquidity[toIndex] = _to;
            ISwerveFi(curve).add_liquidity(liquidity, 0);
        }

        // We want to get back swrv
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            // 4.5% rewards gets sent to treasury
            IERC20(want).safeTransfer(
                IController(controller).comAddr(),
                _want.mul(performanceFee).div(performanceMax)
            );

            deposit();
        }
    }

    // Emergency function call
    function execute(address _target, bytes memory _data)
    public
    payable
    returns (bytes memory response)// knownsec  The hedge function timelock address usage
        {
        require(msg.sender == timelock, "!timelock");

        require(_target != address(0), "!target");

        // call contract in current context
        assembly {
            let succeeded := delegatecall(
            sub(gas(), 5000),
            _target,
            add(_data, 0x20),
            mload(_data),
            0,
            0
            )
```

Knownsec

```
            let size := returndatasize()

            response := mload(0x40)
            mstore(
            0x40,
            add(response, and(add(add(size, 0x20), 0x1f), not(0x1f)))
            )
            mstore(response, size)
            returndatacopy(add(response, 0x20), 0, size)

            switch iszero(succeeded)
            case 1 {
            // throw if delegatecall failed
                revert(add(response, 0x20), size)
            }
        }
    }

    /**
    * Creates a Swerve lock
    */
    function createLock(address lockToken, address escrow, uint256 value, uint256 unlockTime) public {
        require(msg.sender == governance, "!governance");
        IERC20(lockToken).safeApprove(escrow, 0);
        IERC20(lockToken).safeApprove(escrow, value);
        VotingEscrow(escrow).create_lock(value, unlockTime);
    }

    /**
    * Checkpoints the Swerve lock balance
    */
    function checkpoint(address _gauge) public {
        require(msg.sender == governance, "!governance");
        Gauge(_gauge).user_checkpoint(address(this));
    }

    /**
    * Increases the lock amount for Swerve
    */
    function increaseAmount(address lockToken, address escrow, uint256 value) public {
        require(msg.sender == governance, "!governance");
        IERC20(lockToken).safeApprove(escrow, 0);
        IERC20(lockToken).safeApprove(escrow, value);
        VotingEscrow(escrow).increase_amount(value);
    }

    /**
    * Increases the unlock time for Swerve
    */
    function increaseUnlockTime(address escrow, uint256 unlock_time) public {
        require(msg.sender == governance, "!governance");
        VotingEscrow(escrow).increase_unlock_time(unlock_time);
    }

    /**
    * Withdraws an expired lock
    */
    function withdrawLock(address lockToken, address escrow) public {
        require(msg.sender == governance, "!governance");
        uint256 balanceBefore = IERC20(lockToken).balanceOf(address(this));
        VotingEscrow(escrow).withdraw();
        uint256 balanceAfter = IERC20(lockToken).balanceOf(address(this));
        if (balanceAfter > balanceBefore) {
            IERC20(lockToken).safeTransfer(msg.sender, balanceAfter.sub(balanceBefore));
        }
    }

    // **** Internal functions ****
    function _swap(
        address _from,
        address _to,
        uint256 _amount
    ) internal {// Knownsec  exchange token
        // Swap with uniswap
        IERC20(_from).safeApprove(univ2Router2, 0);
        IERC20(_from).safeApprove(univ2Router2, _amount);

        address[] memory path;

        if (_from == weth || _to == weth) {
            path = new address[](2);
            path[0] = _from;
            path[1] = _to;
        } else {
            path = new address[](3);
            path[0] = _from;
            path[1] = weth;
```

knownsec

```
            path[2] = _to;
        }

        UniswapRouterV2(univ2Router2).swapExactTokensForTokens(
            _amount,
            0,
            path,
            address(this),
            now.add(60)
        );
    }
}
```

***StrategyUniEthDaiLp.sol***

```
pragma solidity ^0.6.2;

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

import "../interface/IController.sol";
import "../interface/IStrategy.sol";
import "../interface/IStakingRewards.sol";
import "../interface/UniswapRouterV2.sol";

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

    // Staking rewards address for ETH/DAI LP providers
    address public constant rewards = 0xa1484C3aa22a66C62b77E0AE78E15258bd0cB711;

    // want eth/dai lp tokens
    address public constant want = 0xA478c2975Ab1Ea89e8196811F51A7B7Ade33eB11;

    // tokens we're farming
    address public constant uni = 0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984;

    // stablecoins
    address public constant dai = 0x6B175474E89094C44Da98b954EedeAC495271d0F;

    // weth
    address public constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    // dex
    address public univ2Router2 = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;

    // Fees 5% in total
    // - 1.5%     keepUNI for development fund
    // - 2%       performanceFee for community fund
    // - 1.5%     used to burn/repurchase btfs
    uint256 public keepUNI = 150;
    uint256 public constant keepUNIMax = 10000;

    uint256 public performanceFee = 200;
    uint256 public constant performanceMax = 10000;

    uint256 public burnFee = 150;
    uint256 public constant burnMax = 10000;

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

    address public governance;
    address public controller;
    address public strategist;
    address public timelock;
    address public btf;

    constructor(
        address _governance,
        address _strategist,
        address _controller,
        address _timelock,
        address _btf
    ) public {
        governance = _governance;
        strategist = _strategist;
        controller = _controller;
        timelock = _timelock;
        btf = _btf;
    }//knownsec//  initialization contract，set governance、strategist、controller、timelock、btf address
```

```
// **** Views ****

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

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

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

function getName() external pure returns (string memory) {
    return "StrategyUniEthDaiLp";
}

function getHarvestable() external view returns (uint256) {
    return IStakingRewards(rewards).earned(address(this));
}

// **** Setters ****

function setKeepUNI(uint256 _keepUNI) external {
    require(msg.sender == governance, "!governance");
    keepUNI = _keepUNI;
}

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

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

function setBurnFee(uint256 _burnFee) external {
    require(msg.sender == governance, "!governance");
    burnFee = _burnFee;
}

function setStrategist(address _strategist) external {
    require(msg.sender == governance, "!governance");
    strategist = _strategist;
}

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

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

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

// **** State Mutations ****

function deposit() public {
    uint256 _want = IERC20(want).balanceOf(address(this));
    if (_want > 0) {
        IERC20(want).safeApprove(rewards, 0);
        IERC20(want).approve(rewards, _want);
        IStakingRewards(rewards).stake(_want);
    }
}

// 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");
    balance = _asset.balanceOf(address(this));
    _asset.safeTransfer(controller, balance);
}

// Contoller only function for withdrawing for free
// This is used to swap between vaults
function freeWithdraw(uint256 _amount) external {
```

Knownsec

```
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }
        IERC20(want).safeTransfer(msg.sender, _amount);
    }

    // Withdraw partial funds, normally used with a vault withdrawal
    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }

        if (withdrawalFee > 0) {
            uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
            IERC20(want).safeTransfer(IController(controller).comAddr(), _fee);
            _amount = _amount.sub(_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);
    }

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

        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 _withdrawAll() internal {
        _withdrawSome(balanceOfPool());
    }

    function _withdrawSome(uint256 _amount) internal returns (uint256) {
        IStakingRewards(rewards).withdraw(_amount);
        return _amount;
    }

    function brine() public {
        harvest();
    }

    function harvest() public {
        // Anyone can harvest it at any given time.
        // I understand the possibility of being frontrun
        // But ETH is a dark forest, and I wanna see how this plays out
        // i.e. will be be heavily frontrunned?
        //          if so, a new strategy will be deployed.

        // Collects UNI tokens
        IStakingRewards(rewards).getReward();
        uint256 _uni = IERC20(uni).balanceOf(address(this));
        if (_uni > 0) {
            // 10% is locked up for future gov
            if (keepUNI > 0) {
                uint256 _keepUNI = _uni.mul(keepUNI).div(keepUNIMax);
                IERC20(uni).safeTransfer(
                    IController(controller).devAddr(),
                    _keepUNI
                );
                _uni = _uni.sub(_keepUNI);
            }

            _swap(uni, weth, _uni);
        }

        // Swap half WETH for DAI
        uint256 _weth = IERC20(weth).balanceOf(address(this));
        if (_weth > 0) {
            // Burn some btfs first
            if (burnFee > 0) {
                uint256 _burnFee = _weth.mul(burnFee).div(burnMax);
```

knownsec

```
                _swap(weth, btf, _burnFee);
                IERC20(btf).transfer(
                    IController(controller).burnAddr(),
                    IERC20(btf).balanceOf(address(this))
                );
                _weth = _weth.sub(_burnFee);
            }

            _swap(weth, dai, _weth.div(2));
        }

        // Adds in liquidity for ETH/DAI
        _weth = IERC20(weth).balanceOf(address(this));
        uint256 _dai = IERC20(dai).balanceOf(address(this));
        if (_weth > 0 && _dai > 0) {
            IERC20(weth).safeApprove(univ2Router2, 0);
            IERC20(weth).safeApprove(univ2Router2, _weth);

            IERC20(dai).safeApprove(univ2Router2, 0);
            IERC20(dai).safeApprove(univ2Router2, _dai);

            UniswapRouterV2(univ2Router2).addLiquidity(
                weth,
                dai,
                _weth,
                _dai,
                0,
                0,
                address(this),
                now + 60
            );

            // Donates DUST
            IERC20(weth).transfer(
                IController(controller).comAddr(),
                IERC20(weth).balanceOf(address(this))
            );
            IERC20(dai).transfer(
                IController(controller).comAddr(),
                IERC20(dai).balanceOf(address(this))
            );
        }

        // We want to get back UNI ETH/DAI LP tokens
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            // Performance fee
            if (performanceFee > 0) {
                IERC20(want).safeTransfer(
                    IController(controller).comAddr(),
                    _want.mul(performanceFee).div(performanceMax)
                );
            }

            deposit();
        }
    }

    // Emergency function call
    function execute(address _target, bytes memory _data)
    public
    payable
    returns (bytes memory response)
    {
        require(msg.sender == timelock, "!timelock");

        require(_target != address(0), "!target");

        // call contract in current context
        assembly {
            let succeeded := delegatecall(
            sub(gas(), 5000),
            _target,
            add(_data, 0x20),
            mload(_data),
            0,
            0
            )
            let size := returndatasize()

            response := mload(0x40)
            mstore(
            0x40,
            add(response, and(add(add(size, 0x20), 0x1f), not(0x1f)))
            )
            mstore(response, size)
            returndatacopy(add(response, 0x20), 0, size)
```

knownsec

```
                switch iszero(succeeded)
                case 1 {
                // throw if delegatecall failed
                    revert(add(response, 0x20), size)
                }
            }
        }

        // **** Internal functions ****
        function _swap(
            address _from,
            address _to,
            uint256 _amount
        ) internal {
            // Swap with uniswap
            IERC20(_from).safeApprove(univ2Router2, 0);
            IERC20(_from).safeApprove(univ2Router2, _amount);

            address[] memory path;

            if (_from == weth || _to == weth) {
                path = new address[](2);
                path[0] = _from;
                path[1] = _to;
            } else {
                path = new address[](3);
                path[0] = _from;
                path[1] = weth;
                path[2] = _to;
            }

            UniswapRouterV2(univ2Router2).swapExactTokensForTokens(
                _amount,
                0,
                path,
                address(this),
                now.add(60)
            );
        }
}
```

***StrategyUniEthUsdcLp.sol***

```
pragma solidity ^0.6.2;

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

import "../interface/IController.sol";
import "../interface/IStrategy.sol";
import "../interface/IStakingRewards.sol";
import "../interface/UniswapRouterV2.sol";

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

    // Staking rewards address for ETH/USDC LP providers
    address public constant rewards = 0x7FBa4B8Dc5E7616e59622806932DBea72537A56b;

    // want eth/usdc lp tokens
    address public constant want = 0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc;

    // tokens we're farming
    address public constant uni = 0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984;

    // stablecoins
    address public constant usdc = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;

    // weth
    address public constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    // dex
    address public univ2Router2 = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;

    // Fees 5% in total
    // - 1.5%     keepUNI for development fund
    // - 2%       performanceFee for community fund
    // - 1.5%     used to burn/repurchase btfs
    uint256 public keepUNI = 150;
    uint256 public constant keepUNIMax = 10000;
```

```
    uint256 public performanceFee = 200;
    uint256 public constant performanceMax = 10000;

    uint256 public burnFee = 150;
    uint256 public constant burnMax = 10000;

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

    address public governance;
    address public controller;
    address public strategist;
    address public timelock;
    address public btf;

    constructor(
        address _governance,
        address _strategist,
        address _controller,
        address _timelock,
        address _btf
    ) public {
        governance = _governance;
        strategist = _strategist;
        controller = _controller;
        timelock = _timelock;
        btf = _btf;
      }

    // **** Views ****

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

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

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

    function getName() external pure returns (string memory) {
        return "StrategyUniEthUsdcLp";
    }

    function getHarvestable() external view returns (uint256) {
        return IStakingRewards(rewards).earned(address(this));
    }

    // **** Setters ****

    function setKeepUNI(uint256 _keepUNI) external {
        require(msg.sender == governance, "!governance");
        keepUNI = _keepUNI;
    }

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

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

    function setBurnFee(uint256 _burnFee) external {
        require(msg.sender == governance, "!governance");
        burnFee = _burnFee;
    }

    function setStrategist(address _strategist) external {
        require(msg.sender == governance, "!governance");
        strategist = _strategist;
    }

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

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

Knownsec

```
    }

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

    // **** State Mutations ****

    function deposit() public {
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            IERC20(want).safeApprove(rewards, 0);
            IERC20(want).approve(rewards, _want);
            IStakingRewards(rewards).stake(_want);
        }
    }

    // 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");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }

    // Contoller only function for withdrawing for free
    // This is used to swap between vaults
    function freeWithdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }
        IERC20(want).safeTransfer(msg.sender, _amount);
    }

    // Withdraw partial funds, normally used with a vault withdrawal
    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }

        if (withdrawalFee > 0) {
            uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
            IERC20(want).safeTransfer(IController(controller).comAddr(), _fee);
            _amount = _amount.sub(_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);
    }

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

        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 _withdrawAll() internal {
        _withdrawSome(balanceOfPool());
    }

    function _withdrawSome(uint256 _amount) internal returns (uint256) {
        IStakingRewards(rewards).withdraw(_amount);
        return _amount;
    }

    function brine() public {
    harvest();
    }

    function harvest() public {
```

Knownsec

```
        // Anyone can harvest it at any given time.
        // I understand the possibility of being frontrun
        // But ETH is a dark forest, and I wanna see how this plays out
        // i.e. will be be heavily frontrunned?
        //          if so, a new strategy will be deployed.

        // Collects UNI tokens
        IStakingRewards(rewards).getReward();
        uint256 _uni = IERC20(uni).balanceOf(address(this));
        if (_uni > 0) {
            if (keepUNI > 0) {
                // 10% is locked up for future gov
                uint256 _keepUNI = _uni.mul(keepUNI).div(keepUNIMax);
                IERC20(uni).safeTransfer(
                    IController(controller).devAddr(),
                    _keepUNI
                );
                _uni = _uni.sub(_keepUNI);
            }

            _swap(uni, weth, _uni);
        }

        // Swap half WETH for USDC
        uint256 _weth = IERC20(weth).balanceOf(address(this));
        if (_weth > 0) {
            // Burn some btfs first
            if (burnFee > 0) {
                uint256 _burnFee = _weth.mul(burnFee).div(burnMax);
                _swap(weth, btf, _burnFee);
                IERC20(btf).transfer(
                    IController(controller).burnAddr(),
                    IERC20(btf).balanceOf(address(this))
                );
                _weth = _weth.sub(_burnFee);
            }

            _swap(weth, usdc, _weth.div(2));
        }

        // Adds in liquidity for ETH/USDC
        _weth = IERC20(weth).balanceOf(address(this));
        uint256 _usdc = IERC20(usdc).balanceOf(address(this));
        if (_weth > 0 && _usdc > 0) {
            IERC20(weth).safeApprove(univ2Router2, 0);
            IERC20(weth).safeApprove(univ2Router2, _weth);

            IERC20(usdc).safeApprove(univ2Router2, 0);
            IERC20(usdc).safeApprove(univ2Router2, _usdc);

            UniswapRouterV2(univ2Router2).addLiquidity(
                weth,
                usdc,
                _weth,
                _usdc,
                0,
                0,
                address(this),
                now + 60
            );

            // Donates DUST
            IERC20(weth).transfer(
                IController(controller).comAddr(),
                IERC20(weth).balanceOf(address(this))
            );
            IERC20(usdc).transfer(
                IController(controller).comAddr(),
                IERC20(usdc).balanceOf(address(this))
            );
        }

        // We want to get back UNI ETH/USDC LP tokens
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            // Performance fee
            if (performanceFee > 0) {
                IERC20(want).safeTransfer(
                    IController(controller).comAddr(),
                    _want.mul(performanceFee).div(performanceMax)
                );
            }

            deposit();
        }
    }
```

knownsec

```
    // Emergency function call
    function execute(address _target, bytes memory _data)
    public
    payable
    returns (bytes memory response)
    {
        require(msg.sender == timelock, "!timelock");

        require(_target != address(0), "!target");

        // call contract in current context
        assembly {
            let succeeded := delegatecall(
            sub(gas(), 5000),
            _target,
            add(_data, 0x20),
            mload(_data),
            0,
            0
            )
            let size := returndatasize()

            response := mload(0x40)
            mstore(
            0x40,
            add(response, and(add(add(size, 0x20), 0x1f), not(0x1f)))
            )
            mstore(response, size)
            returndatacopy(add(response, 0x20), 0, size)

            switch iszero(succeeded)
            case 1 {
            // throw if delegatecall failed
                revert(add(response, 0x20), size)
            }
        }
    }

    // **** Internal functions ****
    function _swap(
        address _from,
        address _to,
        uint256 _amount
      ) internal {
        // Swap with uniswap
        IERC20(_from).safeApprove(univ2Router2, 0);
        IERC20(_from).safeApprove(univ2Router2, _amount);

        address[] memory path;

        if (_from == weth || _to == weth) {
            path = new address[](2);
            path[0] = _from;
            path[1] = _to;
        } else {
            path = new address[](3);
            path[0] = _from;
            path[1] = weth;
            path[2] = _to;
        }

        UniswapRouterV2(univ2Router2).swapExactTokensForTokens(
            _amount,
            0,
            path,
            address(this),
            now.add(60)
        );
    }
}
```

***StrategyUniEthUsdtLp.sol***

```
pragma solidity ^0.6.2;

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

import "../interface/IController.sol";
import "../interface/IStrategy.sol";
import "../interface/IStakingRewards.sol";
import "../interface/UniswapRouterV2.sol";
import "../interface/USDT.sol";
```

Knownsec

```
contract StrategyUniEthUsdtLp {
    // v2 Uses uniswap for less gas
    // We can roll back to v1 if the liquidity is there

    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    // Staking rewards address for ETH/USDT LP providers
    address
    public constant rewards = 0x6C3e4cb2E96B01F4b866965A91ed4437839A121a;

    // want eth/usdt lp tokens
    address public constant want = 0x0d4a11d5EEaaC28EC3F61d100daF4d40471f1852;

    // tokens we're farming
    address public constant uni = 0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984;

    // stablecoins
    address public constant usdt = 0xdAC17F958D2ee523a2206206994597C13D831ec7;

    // weth
    address public constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    // dex
    address public univ2Router2 = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;

    // Fees 5% in total
    // - 1.5%    keepUNI for development fund
    // - 2%      performanceFee for community fund
    // - 1.5%    used to burn/repurchase btfs
    uint256 public keepUNI = 150;
    uint256 public constant keepUNIMax = 10000;

    uint256 public performanceFee = 200;
    uint256 public constant performanceMax = 10000;

    uint256 public burnFee = 150;
    uint256 public constant burnMax = 10000;

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

    address public governance;
    address public controller;
    address public strategist;
    address public timelock;
    address public btf;

    constructor(
        address _governance,
        address _strategist,
        address _controller,
        address _timelock,
        address _btf
    ) public {
        governance = _governance;
        strategist = _strategist;
        controller = _controller;
        timelock = _timelock;
        btf = _btf;
    }

    // **** Views ****

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

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

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

    function getName() external pure returns (string memory) {
        return "StrategyUniEthUsdtLp";
    }

    function getHarvestable() external view returns (uint256) {
        return IStakingRewards(rewards).earned(address(this));
    }

    // **** Setters ****
```

Knownsec

```
function setKeepUNI(uint256 _keepUNI) external {
        require(msg.sender == governance, "!governance");
        keepUNI = _keepUNI;
}

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

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

function setBurnFee(uint256 _burnFee) external {
        require(msg.sender == governance, "!governance");
        burnFee = _burnFee;
}

function setStrategist(address _strategist) external {
        require(msg.sender == governance, "!governance");
        strategist = _strategist;
}

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

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

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

// **** State Mutations ****

function deposit() public {
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
                IERC20(want).safeApprove(rewards, 0);
                IERC20(want).approve(rewards, _want);
                IStakingRewards(rewards).stake(_want);
        }
}

// Contoller only function for withdrawing for free
// This is used to swap between vaults
function freeWithdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
                _amount = _withdrawSome(_amount.sub(_balance));
                _amount = _amount.add(_balance);
        }
        IERC20(want).safeTransfer(msg.sender, _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");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
}

// Withdraw partial funds, normally used with a vault withdrawal
function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
                _amount = _withdrawSome(_amount.sub(_balance));
                _amount = _amount.add(_balance);
        }

        if (withdrawalFee > 0) {
                uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
                IERC20(want).safeTransfer(IController(controller).comAddr(), _fee);
                _amount = _amount.sub(_fee);
        }
```

Knownsec

```
        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);
    }

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

        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 _withdrawAll() internal {
        _withdrawSome(balanceOfPool());
    }

    function _withdrawSome(uint256 _amount) internal returns (uint256) {
        IStakingRewards(rewards).withdraw(_amount);
        return _amount;
    }

    function brine() public {
        harvest();
    }

    function harvest() public {
        // Anyone can harvest it at any given time.
        // I understand the possibility of being frontrun
        // But ETH is a dark forest, and I wanna see how this plays out
        // i.e. will be be heavily frontrunned?
        //          if so, a new strategy will be deployed.

        // Collects UNI tokens
        IStakingRewards(rewards).getReward();
        uint256 _uni = IERC20(uni).balanceOf(address(this));
        if (_uni > 0) {
            // 10% is locked up for future gov
            if (keepUNI > 0) {
                uint256 _keepUNI = _uni.mul(keepUNI).div(keepUNIMax);
                IERC20(uni).safeTransfer(
                    IController(controller).devAddr(),
                    _keepUNI
                );
                _uni = _uni.sub(_keepUNI);
            }

            _swap(uni, weth, _uni);
        }

        // Swap half WETH for USDT
        uint256 _weth = IERC20(weth).balanceOf(address(this));
        if (_weth > 0) {
            // Burn some btfs first
            if (burnFee > 0) {
                uint256 _burnFee = _weth.mul(burnFee).div(burnMax);
                _swap(weth, btf, _burnFee);
                IERC20(btf).transfer(
                    IController(controller).burnAddr(),
                    IERC20(btf).balanceOf(address(this))
                );
                _weth = _weth.sub(_burnFee);
            }

            _swap(weth, usdt, _weth.div(2));
        }

        // Adds in liquidity for ETH/usdt
        _weth = IERC20(weth).balanceOf(address(this));
        uint256 _usdt = IERC20(usdt).balanceOf(address(this));
        if (_weth > 0 && _usdt > 0) {
            IERC20(weth).safeApprove(univ2Router2, 0);
            IERC20(weth).safeApprove(univ2Router2, _weth);

            USDT(usdt).approve(univ2Router2, _usdt);

            UniswapRouterV2(univ2Router2).addLiquidity(
                weth,
                usdt,
```

knownsec

```
                _weth,
                _usdt,
                0,
                0,
                address(this),
                now + 60
            );

            // Donates DUST
            IERC20(weth).transfer(
                IController(controller).comAddr(),
                IERC20(weth).balanceOf(address(this))
            );
            USDT(usdt).transfer(
                IController(controller).comAddr(),
                IERC20(usdt).balanceOf(address(this))
            );
        }

        // We want to get back UNI ETH/usdt LP tokens
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            // Performance fee
            if (performanceFee > 0) {
                IERC20(want).safeTransfer(
                    IController(controller).comAddr(),
                    _want.mul(performanceFee).div(performanceMax)
                );
            }

            deposit();
        }
    }

    // Emergency function call
    function execute(address _target, bytes memory _data)
    public
    payable
    returns (bytes memory response)
     {
        require(msg.sender == timelock, "!timelock");

        require(_target != address(0), "!target");

        // call contract in current context
        assembly {
            let succeeded := delegatecall(
            sub(gas(), 5000),
            _target,
            add(_data, 0x20),
            mload(_data),
            0,
            0
            )
            let size := returndatasize()

            response := mload(0x40)
            mstore(
            0x40,
            add(response, and(add(add(size, 0x20), 0x1f), not(0x1f)))
            )
            mstore(response, size)
            returndatacopy(add(response, 0x20), 0, size)

            switch iszero(succeeded)
            case 1 {
            // throw if delegatecall failed
                revert(add(response, 0x20), size)
            }
        }
    }

    // **** Internal functions ****

    function _swap(
        address _from,
        address _to,
        uint256 _amount
      ) internal {
        // Swap with uniswap
        IERC20(_from).safeApprove(univ2Router2, 0);
        IERC20(_from).safeApprove(univ2Router2, _amount);

        address[] memory path;

        if (_from == weth || _to == weth) {
            path = new address[](2);
```

knownsec

```
            path[0] = _from;
            path[1] = _to;
        } else {
            path = new address[](3);
            path[0] = _from;
            path[1] = weth;
            path[2] = _to;
        }

        UniswapRouterV2(univ2Router2).swapExactTokensForTokens(
            amount,
            0,
            path,
            address(this),
            now.add(60)
        );
    }
}
```

***StrategyUniEthWbtcLp.sol***

```
pragma solidity ^0.6.2;

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

import "../interface/IController.sol";
import "../interface/IStrategy.sol";
import "../interface/IStakingRewards.sol";
import "../interface/UniswapRouterV2.sol";

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

    // Staking rewards address for ETH/WBTC LP providers
    address
    public constant rewards = 0xCA35e32e7926b96A9988f61d510E038108d8068e;

    // want eth/wbtc lp tokens
    address public constant want = 0xBb2b8038a1640196FbE3e38816F3e67Cba72D940;

    // tokens we're farming
    address public constant uni = 0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984;

    // stablecoins
    address public constant wbtc = 0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599;

    // weth
    address public constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    // dex
    address public univ2Router2 = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;

    // Fees 5% in total
    // - 1.5%     keepUNI for development fund
    // - 2%       performanceFee for community fund
    // - 1.5%     used to burn/repurchase btfs
    uint256 public keepUNI = 150;
    uint256 public constant keepUNIMax = 10000;

    uint256 public performanceFee = 200;
    uint256 public constant performanceMax = 10000;

    uint256 public burnFee = 150;
    uint256 public constant burnMax = 10000;

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

    address public governance;
    address public controller;
    address public strategist;
    address public timelock;
    address public btf;

    constructor(
        address _governance,
        address _strategist,
        address _controller,
        address _timelock,
        address _btf
```

knownsec

```
) public {
        governance = _governance;
        strategist = _strategist;
        controller = _controller;
        timelock = _timelock;
        btf = _btf;
}

 // **** Views ****

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

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

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

function getName() external pure returns (string memory) {
        return "StrategyUniEthWbtcLp";
}

function getHarvestable() external view returns (uint256) {
        return IStakingRewards(rewards).earned(address(this));
}

// **** Setters ****

function setKeepUNI(uint256 _keepUNI) external {
        require(msg.sender == governance, "!governance");
        keepUNI = _keepUNI;
}

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

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

function setBurnFee(uint256 _burnFee) external {
        require(msg.sender == governance, "!governance");
        burnFee = _burnFee;
}

function setStrategist(address _strategist) external {
        require(msg.sender == governance, "!governance");
        strategist = _strategist;
}

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

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

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

// **** State Mutations ****

function deposit() public {
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
                IERC20(want).safeApprove(rewards, 0);
                IERC20(want).approve(rewards, _want);
                IStakingRewards(rewards).stake(_want);
        }
}

// 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");
```

knownsec

```
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }

    // Contoller only function for withdrawing for free
    // This is used to swap between vaults
    function freeWithdraw(uint256 _amount) external {// knownsec  仓库间代币置换
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }
        IERC20(want).safeTransfer(msg.sender, _amount);
    }

    // Withdraw partial funds, normally used with a vault withdrawal
    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }

        if (withdrawalFee > 0) {
            uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
            IERC20(want).safeTransfer(IController(controller).comAddr(), _fee);
            _amount = _amount.sub(_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);
    }

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

        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 _withdrawAll() internal {
        _withdrawSome(balanceOfPool());
    }

    function _withdrawSome(uint256 _amount) internal returns (uint256) {
        IStakingRewards(rewards).withdraw(_amount);
        return _amount;
    }

    function brine() public {
        harvest();
    }

    function harvest() public {
        // Anyone can harvest it at any given time.
        // I understand the possibility of being frontrun
        // But ETH is a dark forest, and I wanna see how this plays out
        // i.e. will be be heavily frontrunned?
        //          if so, a new strategy will be deployed.

        // Collects UNI tokens
        IStakingRewards(rewards).getReward();
        uint256 _uni = IERC20(uni).balanceOf(address(this));
        if (_uni > 0) {
            // 10% is locked up for future gov
            if (keepUNI > 0) {
                uint256 _keepUNI = _uni.mul(keepUNI).div(keepUNIMax);
                IERC20(uni).safeTransfer(
                    IController(controller).devAddr(),
                    _keepUNI
                );
                _uni = _uni.sub(_keepUNI);
            }
            _swap(uni, weth, _uni);
        }
```

Knownsec

```
        // Swap half WETH for WBTC
        uint256 _weth = IERC20(weth).balanceOf(address(this));
        if (_weth > 0) {
            // Burn some btfs first
            if (burnFee > 0) {
                uint256 _burnFee = _weth.mul(burnFee).div(burnMax);
                _swap(weth, btf, _burnFee);
                IERC20(btf).transfer(
                    IController(controller).burnAddr(),
                    IERC20(btf).balanceOf(address(this))
                );
                _weth = _weth.sub(_burnFee);
            }

            _swap(weth, wbtc, _weth.div(2));
        }

        // Adds in liquidity for ETH/WBTC
        _weth = IERC20(weth).balanceOf(address(this));
        uint256 _wbtc = IERC20(wbtc).balanceOf(address(this));
        if (_weth > 0 && _wbtc > 0) {
            IERC20(weth).safeApprove(univ2Router2, 0);
            IERC20(weth).safeApprove(univ2Router2, _weth);

            IERC20(wbtc).safeApprove(univ2Router2, 0);
            IERC20(wbtc).safeApprove(univ2Router2, _wbtc);

            UniswapRouterV2(univ2Router2).addLiquidity(
                weth,
                wbtc,
                _weth,
                _wbtc,
                0,
                0,
                address(this),
                now + 60
            );

            // Donates DUST
            IERC20(weth).transfer(
                IController(controller).comAddr(),
                IERC20(weth).balanceOf(address(this))
            );
            IERC20(wbtc).transfer(
                IController(controller).comAddr(),
                IERC20(wbtc).balanceOf(address(this))
            );
        }

        // We want to get back UNI ETH/WBTC LP tokens
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            // Performance fee
            if (performanceFee > 0) {
                IERC20(want).safeTransfer(
                    IController(controller).comAddr(),
                    _want.mul(performanceFee).div(performanceMax)
                );
            }

            deposit();
        }
    }

    // Emergency function call
    function execute(address _target, bytes memory _data)
    public
    payable
    returns (bytes memory response)
     {
        require(msg.sender == timelock, "!timelock");

        require(_target != address(0), "!target");

        // call contract in current context
        assembly {
            let succeeded := delegatecall(
            sub(gas(), 5000),
            _target,
            add(_data, 0x20),
            mload(_data),
            0,
            0
            )
            let size := returndatasize()

            response := mload(0x40)
```

Knownsec

```
                mstore(
                0x40,
                add(response, and(add(add(size, 0x20), 0x1f), not(0x1f)))
                )
                mstore(response, size)
                returndatacopy(add(response, 0x20), 0, size)

                switch iszero(succeeded)
                case 1 {
                // throw if delegatecall failed
                      revert(add(response, 0x20), size)
                }
            }
        }

        // **** Internal functions ****
        function _swap(
            address _from,
            address _to,
            uint256 _amount
        ) internal {
            // Swap with uniswap
            IERC20(_from).safeApprove(univ2Router2, 0);
            IERC20(_from).safeApprove(univ2Router2, _amount);

            address[] memory path;

            if (_from == weth || _to == weth) {
                path = new address[](2);
                path[0] = _from;
                path[1] = _to;
            } else {
                path = new address[](3);
                path[0] = _from;
                path[1] = weth;
                path[2] = _to;
            }

            UniswapRouterV2(univ2Router2).swapExactTokensForTokens(
                _amount,
                0,
                path,
                address(this),
                now.add(60)
            );
        }
}
```

***StrategyYfvDai.sol***

```
pragma solidity ^0.6.2;

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

import "../interface/IController.sol";
import "../interface/IStrategy.sol";
import "../interface/IStakingRewards.sol";
import "../interface/UniswapRouterV2.sol";
import "../interface/IYfvRewards.sol";

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

    // Staking rewards address for dai LP providers
    address public constant rewards = 0xC2D55CE14a8e04AEF9B6bCfD105079b63C6a0AC8;

    // want dai stablecoins
    address public constant want = 0x6B175474E89094C44Da98b954EedeAC495271d0F;

    // tokens we're farming
    address public constant yfv = 0x45f24BaEef268BB6d63AEe5129015d69702BCDfa;

    // weth
    address public constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    // yfv vUSD
    address public vUSD = 0x1B8E12F839BD4e73A47adDF76cF7F0097d74c14C;

    // yfv vETH
    address public vETH = 0x76A034e76Aa835363056dd418611E4f81870f16e;
```

```
// dex
address public univ2Router2 = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;

// Fees 5% in total
// - 1.5%     keepYFV for development fund
// - 2%       performanceFee for community fund
// - 1.5%     used to burn/repurchase btfs
uint256 public keepYFV = 150;
uint256 public constant keepYFVMax = 10000;

uint256 public performanceFee = 200;
uint256 public constant performanceMax = 10000;

uint256 public burnFee = 150;
uint256 public constant burnMax = 10000;

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

address public governance;
address public controller;
address public strategist;
address public timelock;
address public btf;

constructor(
        address _governance,
        address _strategist,
        address _controller,
        address _timelock,
        address _btf
) public {
        governance = _governance;
        strategist = _strategist;
        controller = _controller;
        timelock = _timelock;
        btf = _btf;
}

// **** Views ****

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

function balanceOfPool() public view returns (uint256) {
        return IYfvRewards(rewards).balanceOf(want, address(this));
}

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

function getName() external pure returns (string memory) {
        return "StrategyYfvDai";
}

function getHarvestable() external view returns (uint256) {
        return IYfvRewards(rewards).earned(address(this));
}

// **** Setters ****

function setKeepYFV(uint256 _keepYFV) external {
        require(msg.sender == governance, "!governance");
        keepYFV = _keepYFV;
}

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

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

function setBurnFee(uint256 _burnFee) external {
        require(msg.sender == governance, "!governance");
        burnFee = _burnFee;
}

function setStrategist(address _strategist) external {
        require(msg.sender == governance, "!governance");
        strategist = _strategist;
```

Knownsec

```
    }

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

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

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

// **** State Mutations ****

    function deposit() public {
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            IERC20(want).safeApprove(rewards, 0);
            IERC20(want).approve(rewards, _want);
            IYfvRewards(rewards).stake(want, _want, IController(controller).comAddr());
        }
    }

    // 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");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }

    // Contoller only function for withdrawing for free
    // This is used to swap between vaults
    function freeWithdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }
        IERC20(want).safeTransfer(msg.sender, _amount);
    }

    // Withdraw partial funds, normally used with a vault withdrawal
    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }

        if (withdrawalFee > 0) {
            uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
            IERC20(want).safeTransfer(IController(controller).comAddr(), _fee);
            _amount = _amount.sub(_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);
    }

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

        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 _withdrawAll() internal {
        _withdrawSome(balanceOfPool());
    }
```

Knownsec

```
function _withdrawSome(uint256 _amount) internal returns (uint256) {
        IYfvRewards(rewards).withdraw(want, _amount);
        return _amount;
    }

    function brine() public {
        harvest();
    }

    function harvest() public {
        // Anyone can harvest it at any given time.
        // I understand the possibility of being frontrun
        // But ETH is a dark forest, and I wanna see how this plays out
        // i.e. will be be heavily frontrunned?
        //          if so, a new strategy will be deployed.

        // Collects YFV tokens
        IYfvRewards(rewards).getReward();
        uint256 _yfv = IERC20(yfv).balanceOf(address(this));
        if (_yfv > 0) {
            if (keepYFV > 0) {
                // some yfv locked up for future gov
                uint256 _keepYFV = _yfv.mul(keepYFV).div(keepYFVMax);
                IERC20(yfv).safeTransfer(
                    IController(controller).devAddr(),
                    _keepYFV
                );
                _yfv = _yfv.sub(_keepYFV);
            }

            if (burnFee > 0) {
                // Burn some btf
                uint256 _burnFee = _yfv.mul(burnFee).div(burnMax);
                _swap(yfv, btf, _burnFee);
                IERC20(btf).transfer(
                    IController(controller).burnAddr(),
                    IERC20(btf).balanceOf(address(this))
                );
                _yfv = _yfv.sub(_burnFee);
            }

            // swap for want
            _swap(yfv, want, _yfv);
        }

        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            // Performance fee
            if (performanceFee > 0) {
                IERC20(want).safeTransfer(
                    IController(controller).comAddr(),
                    _want.mul(performanceFee).div(performanceMax)
                );
            }

            deposit();
        }
    }

    // Emergency function call
    function execute(address _target, bytes memory _data)
    public
    payable
    returns (bytes memory response)
     {
        require(msg.sender == timelock, "!timelock");

        require(_target != address(0), "!target");

        // call contract in current context
        assembly {
            let succeeded := delegatecall(
            sub(gas(), 5000),
            _target,
            add(_data, 0x20),
            mload(_data),
            0,
            0
            )
            let size := returndatasize()

            response := mload(0x40)
            mstore(
            0x40,
            add(response, and(add(add(size, 0x20), 0x1f), not(0x1f)))
            )
            mstore(response, size)
```

Knownsec

```
                returndatacopy(add(response, 0x20), 0, size)

                switch iszero(succeeded)
                case 1 {
                // throw if delegatecall failed
                    revert(add(response, 0x20), size)
                }
            }
        }

        // **** Internal functions ****
        function _swap(
            address _from,
            address _to,
            uint256 _amount
        ) internal {
            // Swap with uniswap
            IERC20(_from).safeApprove(univ2Router2, 0);
            IERC20(_from).safeApprove(univ2Router2, _amount);

            address[] memory path;

            if (_from == weth || _to == weth) {
                path = new address[](2);
                path[0] = _from;
                path[1] = _to;
            } else {
                path = new address[](3);
                path[0] = _from;
                path[1] = weth;
                path[2] = _to;
            }

            UniswapRouterV2(univ2Router2).swapExactTokensForTokens(
                _amount,
                0,
                path,
                address(this),
                now.add(60)
            );
        }
}
```

***StrategyYfvTusd.sol***

```
pragma solidity ^0.6.2;

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

import "../interface/IController.sol";
import "../interface/IStrategy.sol";
import "../interface/IStakingRewards.sol";
import "../interface/UniswapRouterV2.sol";
import "../interface/IYfvRewards.sol";

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

    // Staking rewards address for tusd LP providers
    address public constant rewards = 0xC2D55CE14a8e04AEF9B6bCfD105079b63C6a0AC8;

    // want tusd stablecoins
    address public constant want = 0x0000000000085d4780B73119b644AE5ecd22b376;

    // tokens we're farming
    address public constant yfv = 0x45f24BaEef268BB6d63AEe5129015d69702BCDfa;

    // weth
    address public constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    // yfv vUSD
    address public vUSD = 0x1B8E12F839BD4e73A47adDF76cF7F0097d74c14C;

    // yfv vETH
    address public vETH = 0x76A034e76Aa835363056dd418611E4f81870f16e;

    // dex
    address public univ2Router2 = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;

    // Fees 5% in total
```

Knownsec

```
// - 1.5%     keepYFV for development fund
// - 2%        performanceFee for community fund
// - 1.5%     used to burn/repurchase btfs
uint256 public keepYFV = 150;
uint256 public constant keepYFVMax = 10000;

uint256 public performanceFee = 200;
uint256 public constant performanceMax = 10000;

uint256 public burnFee = 150;
uint256 public constant burnMax = 10000;

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

address public governance;
address public controller;
address public strategist;
address public timelock;
address public btf;

constructor(
        address _governance,
        address _strategist,
        address _controller,
        address _timelock,
        address _btf
) public {
        governance = _governance;
        strategist = _strategist;
        controller = _controller;
        timelock = _timelock;
        btf = _btf;
}

// **** Views ****

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

function balanceOfPool() public view returns (uint256) {
        return IYfvRewards(rewards).balanceOf(want, address(this));
}

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

function getName() external pure returns (string memory) {
        return "StrategyYfvTusd";
}

function getHarvestable() external view returns (uint256) {
        return IYfvRewards(rewards).earned(address(this));
}

// **** Setters ****

function setKeepYFV(uint256 _keepYFV) external {
        require(msg.sender == governance, "!governance");
        keepYFV = _keepYFV;
}

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

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

function setBurnFee(uint256 _burnFee) external {
        require(msg.sender == governance, "!governance");
        burnFee = _burnFee;
}

function setStrategist(address _strategist) external {
        require(msg.sender == governance, "!governance");
        strategist = _strategist;
}

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

Knownsec

```
    }

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

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

    // **** State Mutations ****

    function deposit() public {
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            IERC20(want).safeApprove(rewards, 0);
            IERC20(want).approve(rewards, _want);
            IYfvRewards(rewards).stake(want, _want, IController(controller).comAddr());
        }
    }

    // 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");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }

    // Contoller only function for withdrawing for free
    // This is used to swap between vaults
    function freeWithdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }
        IERC20(want).safeTransfer(msg.sender, _amount);
    }

    // Withdraw partial funds, normally used with a vault withdrawal
    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }

        if (withdrawalFee > 0) {
            uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
            IERC20(want).safeTransfer(IController(controller).comAddr(), _fee);
            _amount = _amount.sub(_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);
    }

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

        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 _withdrawAll() internal {
        _withdrawSome(balanceOfPool());
    }

    function _withdrawSome(uint256 _amount) internal returns (uint256) {
        IYfvRewards(rewards).withdraw(want, _amount);
        return _amount;
    }
```

Knownsec

```
function brine() public {
        harvest();
    }

    function harvest() public {
        // Anyone can harvest it at any given time.
        // I understand the possibility of being frontrun
        // But ETH is a dark forest, and I wanna see how this plays out
        // i.e. will be be heavily frontrunned?
        //          if so, a new strategy will be deployed.

        // Collects YFV tokens
        IYfvRewards(rewards).getReward();
        uint256 _yfv = IERC20(yfv).balanceOf(address(this));
        if (_yfv > 0) {
            if (keepYFV > 0) {
                // some yfv locked up for future gov
                uint256 _keepYFV = _yfv.mul(keepYFV).div(keepYFVMax);
                IERC20(yfv).safeTransfer(
                    IController(controller).devAddr(),
                    _keepYFV
                );
                _yfv = _yfv.sub(_keepYFV);
            }

            if (burnFee > 0) {
                // Burn some btf
                uint256 _burnFee = _yfv.mul(burnFee).div(burnMax);
                _swap(yfv, btf, _burnFee);
                IERC20(btf).transfer(
                    IController(controller).burnAddr(),
                    IERC20(btf).balanceOf(address(this))
                );
                _yfv = _yfv.sub(_burnFee);
            }

            // swap for want
            _swap(yfv, want, _yfv);
        }

        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            // Performance fee
            if (performanceFee > 0) {
                IERC20(want).safeTransfer(
                    IController(controller).comAddr(),
                    _want.mul(performanceFee).div(performanceMax)
                );
            }

            deposit();
        }
    }

    // Emergency function call
    function execute(address _target, bytes memory _data)
    public
    payable
    returns (bytes memory response)
    {
        require(msg.sender == timelock, "!timelock");

        require(_target != address(0), "!target");

        // call contract in current context
        assembly {
            let succeeded := delegatecall(
            sub(gas(), 5000),
            _target,
            add(_data, 0x20),
            mload(_data),
            0,
            0
            )
            let size := returndatasize()

            response := mload(0x40)
            mstore(
            0x40,
            add(response, and(add(add(size, 0x20), 0x1f), not(0x1f)))
            )
            mstore(response, size)
            returndatacopy(add(response, 0x20), 0, size)

            switch iszero(succeeded)
            case 1 {
            // throw if delegatecall failed
```

Knownsec

```
                    revert(add(response, 0x20), size)
                }
            }
        }

        // **** Internal functions ****
        function _swap(
            address _from,
            address _to,
            uint256 _amount
        ) internal {
            // Swap with uniswap
            IERC20(_from).safeApprove(univ2Router2, 0);
            IERC20(_from).safeApprove(univ2Router2, _amount);

            address[] memory path;

            if (_from == weth || _to == weth) {
                path = new address[](2);
                path[0] = _from;
                path[1] = _to;
            } else {
                path = new address[](3);
                path[0] = _from;
                path[1] = weth;
                path[2] = _to;
            }

            UniswapRouterV2(univ2Router2).swapExactTokensForTokens(
                _amount,
                0,
                path,
                address(this),
                now.add(60)
            );
        }
}
```

***StrategyYfvUsdc.sol***

```
pragma solidity ^0.6.2;

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

import "../interface/IController.sol";
import "../interface/IStrategy.sol";
import "../interface/IStakingRewards.sol";
import "../interface/UniswapRouterV2.sol";
import "../interface/IYfvRewards.sol";

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

    // Staking rewards address for usdc LP providers
    address public constant rewards = 0xC2D55CE14a8e04AEF9B6bCfD105079b63C6a0AC8;

    // want usdc stablecoins
    address public constant want = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;

    // tokens we're farming
    address public constant yfv = 0x45f24BaEef268BB6d63AEe5129015d69702BCDfa;

    // weth
    address public constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    // yfv vUSD
    address public vUSD = 0x1B8E12F839BD4e73A47adDF76cF7F0097d74c14C;

    // yfv vETH
    address public vETH = 0x76A034e76Aa835363056dd418611E4f81870f16e;

    // dex
    address public univ2Router2 = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;

    // Fees 5% in total
    // - 1.5%     keepYFV for development fund
    // - 2%       performanceFee for community fund
    // - 1.5%     used to burn/repurchase btfs
    uint256 public keepYFV = 150;
```

Knownsec

```
    uint256 public constant keepYFVMax = 10000;

    uint256 public performanceFee = 200;
    uint256 public constant performanceMax = 10000;

    uint256 public burnFee = 150;
    uint256 public constant burnMax = 10000;

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

    address public governance;
    address public controller;
    address public strategist;
    address public timelock;
    address public btf;

    constructor(
        address _governance,
        address _strategist,
        address _controller,
        address _timelock,
        address _btf
    ) public {
        governance = _governance;
        strategist = _strategist;
        controller = _controller;
        timelock = _timelock;
        btf = _btf;
    }

    // **** Views ****

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

    function balanceOfPool() public view returns (uint256) {
        return IYfvRewards(rewards).balanceOf(want, address(this));
    }

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

    function getName() external pure returns (string memory) {
        return "StrategyYfvUsdc";
    }

    function getHarvestable() external view returns (uint256) {
        return IYfvRewards(rewards).earned(address(this));
    }

    // **** Setters ****

    function setKeepYFV(uint256 _keepYFV) external {
        require(msg.sender == governance, "!governance");
        keepYFV = _keepYFV;
    }

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

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

    function setBurnFee(uint256 _burnFee) external {
        require(msg.sender == governance, "!governance");
        burnFee = _burnFee;
    }

    function setStrategist(address _strategist) external {
        require(msg.sender == governance, "!governance");
        strategist = _strategist;
    }

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

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

Knownsec

```
        timelock = _timelock;
    }

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

    // **** State Mutations ****

    function deposit() public {
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            IERC20(want).safeApprove(rewards, 0);
            IERC20(want).approve(rewards, _want);
            IYfvRewards(rewards).stake(want, _want, IController(controller).comAddr());
        }
    }

    // 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");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }

    // Contoller only function for withdrawing for free
    // This is used to swap between vaults
    function freeWithdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }
        IERC20(want).safeTransfer(msg.sender, _amount);
    }

    // Withdraw partial funds, normally used with a vault withdrawal
    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }

        if (withdrawalFee > 0) {
            uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
            IERC20(want).safeTransfer(IController(controller).comAddr(), _fee);
            _amount = _amount.sub(_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);
    }

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

        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 _withdrawAll() internal {
        _withdrawSome(balanceOfPool());
    }

    function _withdrawSome(uint256 _amount) internal returns (uint256) {
        IYfvRewards(rewards).withdraw(want, _amount);
        return _amount;
    }

    function brine() public {
        harvest();
    }
```

Knownsec

```
function harvest() public {
        // Anyone can harvest it at any given time.
        // I understand the possibility of being frontrun
        // But ETH is a dark forest, and I wanna see how this plays out
        // i.e. will be be heavily frontrunned?
        //          if so, a new strategy will be deployed.

        // Collects YFV tokens
        IYfvRewards(rewards).getReward();
        uint256 _yfv = IERC20(yfv).balanceOf(address(this));
        if (_yfv > 0) {
            if (keepYFV > 0) {
                // some yfv locked up for future gov
                uint256 _keepYFV = _yfv.mul(keepYFV).div(keepYFVMax);
                IERC20(yfv).safeTransfer(
                    IController(controller).devAddr(),
                    _keepYFV
                );
                _yfv = _yfv.sub(_keepYFV);
            }

            if (burnFee > 0) {
                // Burn some btf
                uint256 _burnFee = _yfv.mul(burnFee).div(burnMax);
                _swap(yfv, btf, _burnFee);
                IERC20(btf).transfer(
                    IController(controller).burnAddr(),
                    IERC20(btf).balanceOf(address(this))
                );
                _yfv = _yfv.sub(_burnFee);
            }

            // swap for want
            _swap(yfv, want, _yfv);
        }

        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            // Performance fee
            if (performanceFee > 0) {
                IERC20(want).safeTransfer(
                    IController(controller).comAddr(),
                    _want.mul(performanceFee).div(performanceMax)
                );
            }

            deposit();
        }
    }

    // Emergency function call
    function execute(address _target, bytes memory _data)
    public
    payable
    returns (bytes memory response)
     {
        require(msg.sender == timelock, "!timelock");

        require(_target != address(0), "!target");

        // call contract in current context
        assembly {
            let succeeded := delegatecall(
            sub(gas(), 5000),
              _target,
            add(_data, 0x20),
            mload(_data),
            0,
            0
            )
            let size := returndatasize()

            response := mload(0x40)
            mstore(
            0x40,
            add(response, and(add(add(size, 0x20), 0x1f), not(0x1f)))
            )
            mstore(response, size)
            returndatacopy(add(response, 0x20), 0, size)

            switch iszero(succeeded)
            case 1 {
            // throw if delegatecall failed
                revert(add(response, 0x20), size)
            }
        }
    }
```

knownsec

```
    // **** Internal functions ****
    function _swap(
        address _from,
        address _to,
        uint256 _amount
    ) internal {
        // Swap with uniswap
        IERC20(_from).safeApprove(univ2Router2, 0);
        IERC20(_from).safeApprove(univ2Router2, _amount);

        address[] memory path;

        if (_from == weth || _to == weth) {
            path = new address[](2);
            path[0] = _from;
            path[1] = _to;
        } else {
            path = new address[](3);
            path[0] = _from;
            path[1] = weth;
            path[2] = _to;
        }

        UniswapRouterV2(univ2Router2).swapExactTokensForTokens(
            _amount,
            0,
            path,
            address(this),
            now.add(60)
        );
    }
}
```

***StrategyYfvUsdt.sol***

```
pragma solidity ^0.6.2;

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

import "../interface/IController.sol";
import "../interface/IStrategy.sol";
import "../interface/IStakingRewards.sol";
import "../interface/UniswapRouterV2.sol";
import "../interface/IYfvRewards.sol";
import "../interface/USDT.sol";

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

    // Staking rewards address for usdt LP providers
    address public constant rewards = 0xC2D55CE14a8e04AEF9B6bCfD105079b63C6a0AC8;

    // want usdt stablecoins
    address public constant want = 0xdAC17F958D2ee523a2206206994597C13D831ec7;

    // tokens we're farming
    address public constant yfv = 0x45f24BaEef268BB6d63AEe5129015d69702BCDfa;

    // weth
    address public constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    // yfv vUSD
    address public vUSD = 0x1B8E12F839BD4e73A47adDF76cF7F0097d74c14C;

    // yfv vETH
    address public vETH = 0x76A034e76Aa835363056dd418611E4f81870f16e;

    // dex
    address public univ2Router2 = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;

    // Fees 5% in total
    // - 1.5%     keepYFV for development fund
    // - 2%       performanceFee for community fund
    // - 1.5%     used to burn/repurchase btfs
    uint256 public keepYFV = 150;
    uint256 public constant keepYFVMax = 10000;

    uint256 public performanceFee = 200;
    uint256 public constant performanceMax = 10000;
```

Knownsec

```
uint256 public burnFee = 150;
uint256 public constant burnMax = 10000;

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

address public governance;
address public controller;
address public strategist;
address public timelock;
address public btf;

constructor(
        address _governance,
        address _strategist,
        address _controller,
        address _timelock,
        address _btf
) public {
        governance = _governance;
        strategist = _strategist;
        controller = _controller;
        timelock = _timelock;
        btf = _btf;
}

// **** Views ****

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

function balanceOfPool() public view returns (uint256) {
        return IYfvRewards(rewards).balanceOf(want, address(this));
}

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

function getName() external pure returns (string memory) {
        return "StrategyYfvUsdt";
}

function getHarvestable() external view returns (uint256) {
        return IYfvRewards(rewards).earned(address(this));
}

// **** Setters ****

function setKeepYFV(uint256 _keepYFV) external {
        require(msg.sender == governance, "!governance");
        keepYFV = _keepYFV;
}

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

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

function setBurnFee(uint256 _burnFee) external {
        require(msg.sender == governance, "!governance");
        burnFee = _burnFee;
}

function setStrategist(address _strategist) external {
        require(msg.sender == governance, "!governance");
        strategist = _strategist;
}

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

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

function setController(address _controller) external {
```

knownsec

```
        require(msg.sender == governance, "!governance");
        controller = _controller;
    }

    // **** State Mutations ****

    function deposit() public {
        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            IERC20(want).safeApprove(rewards, 0);
            IERC20(want).approve(rewards, _want);
            IYfvRewards(rewards).stake(want, _want, IController(controller).comAddr());
        }
    }

    // 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");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }

    // Contoller only function for withdrawing for free
    // This is used to swap between vaults
    function freeWithdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }
        IERC20(want).safeTransfer(msg.sender, _amount);
    }

    // Withdraw partial funds, normally used with a vault withdrawal
    function withdraw(uint256 _amount) external {
        require(msg.sender == controller, "!controller");
        uint256 _balance = IERC20(want).balanceOf(address(this));
        if (_balance < _amount) {
            _amount = _withdrawSome(_amount.sub(_balance));
            _amount = _amount.add(_balance);
        }

        if (withdrawalFee > 0) {
            uint256 _fee = _amount.mul(withdrawalFee).div(withdrawalMax);
            IERC20(want).safeTransfer(IController(controller).comAddr(), _fee);
            _amount = _amount.sub(_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);
    }

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

        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 _withdrawAll() internal {
        _withdrawSome(balanceOfPool());
    }

    function _withdrawSome(uint256 _amount) internal returns (uint256) {
        IYfvRewards(rewards).withdraw(want, _amount);
        return _amount;
    }

    function brine() public {
        harvest();
    }

    function harvest() public {
        // Anyone can harvest it at any given time.
        // I understand the possibility of being frontrun
        // But ETH is a dark forest, and I wanna see how this plays out
```

knownsec

```
        // i.e. will be be heavily frontrunned?
        //          if so, a new strategy will be deployed.

        // Collects YFV tokens
        IYfvRewards(rewards).getReward();
        uint256 _yfv = IERC20(yfv).balanceOf(address(this));
        if (_yfv > 0) {
            if (keepYFV > 0) {
                // some yfv locked up for future gov
                uint256 _keepYFV = _yfv.mul(keepYFV).div(keepYFVMax);
                IERC20(yfv).safeTransfer(
                    IController(controller).devAddr(),
                    _keepYFV
                );
                _yfv = _yfv.sub(_keepYFV);
            }

            if (burnFee > 0) {
                // Burn some btf
                uint256 _burnFee = _yfv.mul(burnFee).div(burnMax);
                _swap(yfv, btf, _burnFee);
                IERC20(btf).transfer(
                    IController(controller).burnAddr(),
                    IERC20(btf).balanceOf(address(this))
                );
                _yfv = _yfv.sub(_burnFee);
            }

            // swap for want
            _swap(yfv, want, _yfv);
        }

        uint256 _want = IERC20(want).balanceOf(address(this));
        if (_want > 0) {
            // Performance fee
            if (performanceFee > 0) {
                IERC20(want).safeTransfer(
                    IController(controller).comAddr(),
                    _want.mul(performanceFee).div(performanceMax)
                );
            }

            deposit();
        }
    }

    // Emergency function call
    function execute(address _target, bytes memory _data)
    public
    payable
    returns (bytes memory response)
     {
        require(msg.sender == timelock, "!timelock");

        require(_target != address(0), "!target");

        // call contract in current context
        assembly {
            let succeeded := delegatecall(
            sub(gas(), 5000),
            _target,
            add(_data, 0x20),
            mload(_data),
            0,
            0
            )
            let size := returndatasize()

            response := mload(0x40)
            mstore(
            0x40,
            add(response, and(add(add(size, 0x20), 0x1f), not(0x1f)))
            )
            mstore(response, size)
            returndatacopy(add(response, 0x20), 0, size)

            switch iszero(succeeded)
            case 1 {
            // throw if delegatecall failed
                revert(add(response, 0x20), size)
            }
        }
    }

    // **** Internal functions ****
    function _swap(
        address _from,
```

Knownsec

```
        address _to,
    uint256 _amount
        ) internal {
        // Swap with uniswap
        IERC20(_from).safeApprove(univ2Router2, 0);
        IERC20(_from).safeApprove(univ2Router2, _amount);

        address[] memory path;

        if (_from == weth || _to == weth) {
            path = new address[](2);
            path[0] = _from;
            path[1] = _to;
        } else {
            path = new address[](3);
            path[0] = _from;
            path[1] = weth;
            path[2] = _to;
        }

        UniswapRouterV2(univ2Router2).swapExactTokensForTokens(
            _amount,
            0,
            path,
            address(this),
            now.add(60)
        );
    }
}
```

***BTFReferral.sol***

```
pragma solidity ^0.6.7;

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

contract BTFReferral {
    mapping(address => address) public referrers; // account_address -> referrer_address
    mapping(address => uint256) public referredCount; // referrer_address -> num_of_referred

    event Referral(address indexed referrer, address indexed farmer);

    // Standard contract ownership transfer.
    address public owner;
    address private nextOwner;

    mapping(address => bool) public isAdmin;//Administrators Arrays

    constructor () public {
    owner = msg.sender;
    }// knownsec  Initialize owner highest privilege

    // Standard modifier on methods invokable only by contract owner.
    modifier onlyOwner {
        require(msg.sender == owner, "OnlyOwner methods called by non-owner.");
        _;
    }

    modifier onlyAdmin {
        require(isAdmin[msg.sender], "OnlyAdmin methods called by non-admin.");
        _;
    }

    // Standard contract ownership transfer implementation,
    function approveNextOwner(address _nextOwner) external onlyOwner {// knownsec  Transfer of ownership rights
Former Owner Approval Former Owner Use
        require(_nextOwner != owner, "Cannot approve current owner.");
        nextOwner = _nextOwner;
    }

    function acceptNextOwner() external {// knownsec  Ownership transfer, New Owner accept, New Owner use, require checksum

        require(msg.sender == nextOwner, "Can only accept preapproved new owner.");
        owner = nextOwner;
    }

    function setReferrer(address farmer, address referrer) public onlyAdmin {
        if (referrers[farmer] == address(0) && referrer != address(0)) {
            referrers[farmer] = referrer;
            referredCount[referrer] += 1;
            emit Referral(referrer, farmer);
        }
    }

    function getReferrer(address farmer) public view returns (address) {
        return referrers[farmer];
```

knownsec

```
    }

    // Set admin status.
    function setAdminStatus(address _admin, bool _status) external onlyOwner {// knownsec  Managing an array of administrators    owner uses
        isAdmin[_admin] = _status;
    }

    event EmergencyERC20Drain(address token, address owner, uint256 amount);

    // owner can drain tokens that are sent here by mistake
    function emergencyERC20Drain(ERC20 token, uint amount) external onlyOwner {// knownsec  Managing an  array  of  administrators,  Emergency transfer out token
        emit EmergencyERC20Drain(address(token), owner, amount);
        token.transfer(owner, amount);
    }
}
```

***BTFToken.sol***

```
pragma solidity 0.6.12;
// SPDX-License-Identifier: MIT

import "@openzeppelin/contracts/GSN/Context.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

// BTFToken with Governance.
contract BTFToken is ERC20("BTFToken", "BTF"), Ownable {
    /// @notice Creates `_amount` token to `_to`. Must only be called by the owner (MasterChef).
  function mint(address _to, uint256 _amount) public onlyOwner {//    knownsec//Issuance of additional tokens
        _mint(_to, _amount);
        _moveDelegates(address(0), _delegates[_to], _amount);
    }

    // Copied and modified from YAM code:
    // https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernanceStorage.sol
    // https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernance.sol
    // Which is copied and modified from COMPOUND:
    // https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/Comp.sol

    /// @notice A record of each accounts delegate
    mapping (address => address) internal _delegates;

    /// @notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint256 votes;
    }

    /// @notice A record of votes checkpoints for each account, by index
    mapping (address => mapping (uint32 => Checkpoint)) public checkpoints;

    /// @notice The number of checkpoints for each account
    mapping (address => uint32) public numCheckpoints;

    /// @notice The EIP-712 typehash for the contract's domain
    bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,address verifyingContract)");

    /// @notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256 nonce,uint256 expiry)");

    /// @notice A record of states for signing / validating signatures
    mapping (address => uint) public nonces;

    /// @notice An event thats emitted when an account changes its delegate
    event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate);

    /// @notice An event thats emitted when a delegate account's vote balance changes
    event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

    /**
     * @notice Delegate votes from `msg.sender` to `delegatee`
     * @param delegator The address to get delegatee for
     */
    function delegates(address delegator)
    external
    view
    returns (address)
    {
```

```
        return _delegates[delegator];
    }

    /**
     * @notice Delegate votes from `msg.sender` to `delegatee`
     * @param delegatee The address to delegate votes to
     */
    function delegate(address delegatee) external {
        return _delegate(msg.sender, delegatee);
    }

    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(
        address delegatee,
        uint nonce,
        uint expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
    external
    {
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name())),
                getChainId(),
                address(this)
            )
        );

        bytes32 structHash = keccak256(
            abi.encode(
                DELEGATION_TYPEHASH,
                delegatee,
                nonce,
                expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "BTF::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "BTF::delegateBySig: invalid nonce");
        require(now <= expiry, "BTF::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }

    /**
     * @notice Gets the current votes balance for `account`
     * @param account The address to get votes balance
     * @return The number of current votes for `account`
     */
    function getCurrentVotes(address account)
    external
    view
    returns (uint256)
    {
        uint32 nCheckpoints = numCheckpoints[account];
        return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
    }

    /**
     * @notice Determine the prior number of votes for an account as of a block number
     * @dev Block number must be a finalized block or else this function will revert to prevent misinformation.
     * @param account The address of the account to check
     * @param blockNumber The block number to get the vote balance at
     * @return The number of votes the account had as of the given block
     */
    function getPriorVotes(address account, uint blockNumber)
    external
    view
```

Knownsec

```
returns (uint256)
{
        require(blockNumber < block.number, "BTF::getPriorVotes: not yet determined");

        uint32 nCheckpoints = numCheckpoints[account];
        if (nCheckpoints == 0) {
            return 0;
        }

        // First check most recent balance
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
            return checkpoints[account][nCheckpoints - 1].votes;
        }

        // Next check implicit zero balance
        if (checkpoints[account][0].fromBlock > blockNumber) {
            return 0;
        }

        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }

    function _delegate(address delegator, address delegatee)// knownsec    Delegate Internal Call
    internal
    {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator); // balance of underlying BTFs (not scaled);
        _delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }

    function _writeCheckpoint(
        address delegatee,
        uint32 nCheckpoints,
        uint256 oldVotes,
        uint256 newVotes
    )
    internal
    {
        uint32 blockNumber = safe32(block.number, "BTF::_writeCheckpoint: block number exceeds 32 bits");

        if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
            checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
        } else {
            checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
            numCheckpoints[delegatee] = nCheckpoints + 1;
        }

        emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
    }
```

Knownsec

```
    function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {
        require(n < 2**32, errorMessage);
        return uint32(n);
    }

    function getChainId() internal pure returns (uint) {
        uint256 chainId;
        assembly { chainId := chainid() }
        return chainId;
    }
}
```

***MasterChef.sol***

```
pragma solidity 0.6.12;

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

import "./BTFToken.sol";
import "./interface/IMigratorChef.sol";
import "./interface/IBTFReferral.sol";

// MasterChef is the master of BTF
contract MasterChef is Ownable {
    uint256 public constant DURATION = 7 days;

    uint256 public TotalSupply = 1000000 * 1e18;
    uint256 public initReward = 100000 * 1e18;
    uint256 public periodFinish = 0;
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    uint256 public rewardRate = 0;

    address public rewardReferral;
    uint256 public constant referralMax = 10000;
    // 1%
    uint256 public referralPercent = 100;

    // Info of each user.
    struct UserInfo {
        uint256 amount;
        uint256 rewardPerTokenPaid;
        uint256 rewards;
    }

    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken;
        uint256 allocPoint;
        uint256 lastUpdateTime;
        uint256 rewardPerTokenStored;
    }

    // The BTF TOKEN!
    BTFToken public btf;
    // The migrator contract. It has a lot of power. Can only be set through governance (owner).
    IMigratorChef public migrator;
    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;
    // Total allocation poitns. Must be the sum of all allocation points in all pools.
    uint256 public totalAllocPoint = 0;
    // The block number when BTF mining starts.
    uint256 public startBlock;
    // add the same LP token only once
    mapping(address => bool) lpExists;

    event RewardAdded(uint256 reward);
    event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
    event RewardPaid(address indexed user, uint256 reward);
    event EmergencyWithdraw(address indexed user, uint256 indexed pid, uint256 amount);

    constructor(BTFToken _btf, uint256 _startBlock) public {
        btf = _btf;
        startBlock = _startBlock;
    }

    function setRewardReferral(address _rewardReferral) external onlyOwner {
```

```
        rewardReferral = _rewardReferral;
    }

    function setReferralPercent(uint256 _referralPercent) external onlyOwner {
        require(_referralPercent > 0 && _referralPercent < referralMax, "_referralPercent is wrong");
        referralPercent = _referralPercent;
    }

    modifier reduceHalve() {
        require(btf.totalSupply() <= TotalSupply, "Out of limited.");

        if (periodFinish == 0) {
            periodFinish = block.timestamp.add(DURATION);
            rewardRate = initReward.div(DURATION);
            btf.mint(address(this), initReward);
        } else if (block.timestamp >= periodFinish) {
            initReward = initReward.sub(initReward.mul(10).div(100));
            rewardRate = initReward.div(DURATION);
            btf.mint(address(this), initReward);
            periodFinish = block.timestamp.add(DURATION);
            emit RewardAdded(initReward);
        }
        _;
    }

    modifier checkStart(){
        require(block.number > startBlock, "not start");
        _;
    }

    modifier updateReward(uint256 _pid, address _user) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];

        pool.rewardPerTokenStored = rewardPerToken(_pid);
        pool.lastUpdateTime = lastTimeRewardApplicable();
        if (_user != address(0)) {
            user.rewards = earned(_pid, _user);
            user.rewardPerTokenPaid = pool.rewardPerTokenStored;
        }
        _;
    }

    function lastTimeRewardApplicable() public view returns (uint256) {
        return (periodFinish == 0 || block.timestamp < periodFinish) ? block.timestamp : periodFinish;
    }

    function rewardPerToken(uint256 _pid) public view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            return pool.rewardPerTokenStored;
        }
        return pool.rewardPerTokenStored.add(
            lastTimeRewardApplicable()
            .sub(pool.lastUpdateTime == 0 ? block.timestamp : pool.lastUpdateTime)
            .mul(rewardRate)
            .mul(pool.allocPoint)
            .div(totalAllocPoint)
            .mul(1e18)
            .div(lpSupply)
        );
    }

    function earned(uint256 _pid, address _user) public view returns (uint256) {
        UserInfo storage user = userInfo[_pid][_user];
        return user.amount
        .mul(rewardPerToken(_pid).sub(user.rewardPerTokenPaid))
        .div(1e18)
        .add(user.rewards);
    }

    // Add a new lp to the pool. Can only be called by the owner.
    function add(uint256 _allocPoint, IERC20 _lpToken) public onlyOwner {
        require(!lpExists[address(_lpToken)], "do not add the same lp token more than once");

        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken : _lpToken,
            allocPoint : _allocPoint,
            lastUpdateTime : 0,
            rewardPerTokenStored : 0
            }));

        lpExists[address(_lpToken)] = true;
    }
```

Knownsec

```
    function poolLength() external view returns (uint256) {
        return poolInfo.length;
    }

    // Update the given pool's BTF allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint) public onlyOwner {
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    // View function to see pending BTFs on frontend.
    function pendingBTF(uint256 _pid, address _user) external view returns (uint256) {
        return earned(_pid, _user);
    }

    function _getReward(uint256 _pid, address _user) private {
        UserInfo storage user = userInfo[_pid][_user];
        uint256 reward = earned(_pid, _user);
        if (reward > 0) {
            user.rewards = 0;

            uint256 btfBal = btf.balanceOf(address(this));
            if (reward > btfBal) {
                reward = btfBal;
            }

            uint256 referrerReward = 0;
            address referrer = address(0);
            if (rewardReferral != address(0)) {
                referrer = IBTFReferral(rewardReferral).getReferrer(_user);
            }
            if (referrer != address(0)) {
                referrerReward = reward.mul(referralPercent).div(referralMax);
                btf.transfer(referrer, referrerReward);
                emit RewardPaid(referrer, referrerReward);
            }

            btf.transfer(_user, reward.sub(referrerReward));
            emit RewardPaid(_user, reward.sub(referrerReward));
        }
    }

    // Deposit LP tokens to MasterChef for BTF allocation.
    function deposit(uint256 _pid, uint256 _amount, address referrer) public updateReward(_pid, msg.sender)
reduceHalve checkStart {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        _getReward(_pid, msg.sender);
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        emit Deposit(msg.sender, _pid, _amount);

        if (rewardReferral != address(0) && referrer != address(0)) {
            IBTFReferral(rewardReferral).setReferrer(msg.sender, referrer);
        }
    }

    // Withdraw LP tokens from MasterChef.
    function withdraw(uint256 _pid, uint256 _amount) public updateReward(_pid, msg.sender) reduceHalve
checkStart {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        _getReward(_pid, msg.sender);
        user.amount = user.amount.sub(_amount);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens & Rewards from MasterChef
    function exit(uint256 _pid) external {
        withdraw(_pid, balanceOf(msg.sender));
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
        user.amount = 0;
        user.rewards = 0;
    }

    // Set the migrator contract. Can only be called by the owner.
    function setMigrator(IMigratorChef _migrator) public onlyOwner {// knownsec  Set the value of migrator
used by the owner. migrator has very high code execution and is associated to 255 lines IERC20 newLpToken =
migrator.migrate(lpToken);
```

```
        migrator = _migrator;
    }

    // Migrate lp token to another lp contract. Can be called by anyone. We trust that migrator contract is good.
    function migrate(uint256 _pid) public {
        require(address(migrator) != address(0), "migrate: no migrator");
        PoolInfo storage pool = poolInfo[_pid];
        IERC20 lpToken = pool.lpToken;
        uint256 bal = lpToken.balanceOf(address(this));
        lpToken.safeApprove(address(migrator), bal);
        IERC20 newLpToken = migrator.migrate(lpToken);
        require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
        pool.lpToken = newLpToken;
    }

}
```

***Migrations.sol***

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.8.0;

contract Migrations {
  address public owner = msg.sender;
  uint public last_completed_migration;

  modifier restricted() {
    require(
      msg.sender == owner,
      "This function is restricted to the contract's owner"
    );
    _;
  }

  function setCompleted(uint completed) public restricted {
    last_completed_migration = completed;
  }
}
```

***MockERC20.sol***

```
pragma solidity 0.6.12;


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


contract MockERC20 is ERC20 {
    constructor(
        string memory name,
        string memory symbol,
        uint256 supply
    ) public ERC20(name, symbol) {
        _mint(msg.sender, supply);
    }
}
```

***Timelock.sol***

```
// COPIED FROM https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/GovernorAlpha.sol
// Copyright 2020 Compound Labs, Inc.
// Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
// 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
// 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
// 3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// Ctrl+f for XXX to see all the modifications.

// XXX: pragma solidity ^0.5.16;
pragma solidity 0.6.12;

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

Knownsec

```
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, uint eta);
    event ExecuteTransaction(bytes32 indexed txHash, address indexed target, uint value, string signature, bytes data, uint eta);
    event QueueTransaction(bytes32 indexed txHash, address indexed target, uint value, string signature, bytes data, uint eta);

    uint public constant GRACE_PERIOD = 14 days;
    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 Timelock.");
        } 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));
```

KNOWNSEC

```
        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
queued.");
        require(getBlockTimestamp() >= eta, "Timelock::executeTransaction: Transaction hasn't surpassed time
lock.");
        require(getBlockTimestamp() <= eta.add(GRACE_PERIOD), "Timelock::executeTransaction:
Transaction is stale.");

        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;
    }
}
```

***Converter.sol***

```
pragma solidity ^0.6.2;

interface Converter {
    function convert(address) external returns (uint256);
}
```

***Gauge.sol***

```
pragma solidity ^0.6.2;

interface Gauge {
    function deposit(uint) external;

    function balanceOf(address) external view returns (uint);

    function withdraw(uint) external;

    function user_checkpoint(address) external;

    function claimable_tokens(address) external returns (uint256);
}

interface VotingEscrow {
    function create_lock(uint256 v, uint256 time) external;

    function increase_amount(uint256 _value) external;

    function increase_unlock_time(uint256 _unlock_time) external;

    function withdraw() external;
}

interface Mintr {
    function mint(address) external;
}
```

***IBTFReferral.sol***

```
pragma solidity ^0.6.2;
```

knownsec

```
interface IBTFReferral {
    function setReferrer(address farmer, address referrer) external;
    function getReferrer(address farmer) external view returns (address);
}
```

***IController.sol***

```
pragma solidity ^0.6.0;

interface IController {
    function vaults(address) external view returns (address);

    function comAddr() external view returns (address);

    function devAddr() external view returns (address);

    function burnAddr() external view returns (address);

    function want(address) external view returns (address); // NOTE: Only StrategyControllerV2 implements this

    function balanceOf(address) external view returns (uint256);

    function withdraw(address, uint256) external;

    function freeWithdraw(address, uint256) external;

    function earn(address, uint256) external;
}
```

***IMigratorChef.sol***

```
pragma solidity 0.6.12;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface IMigratorChef {
    // Perform LP token migration from legacy UniswapV2 to BTFSwap.
    // Take the current LP token address and return the new LP token address.
    // Migrator should have full access to the caller's LP token.
    // Return the new LP token address.
    //
    // XXX Migrator must have allowance access to UniswapV2 LP tokens.
    // BTFSwap must mint EXACTLY the same amount of BTFSwap LP tokens or
    // else something bad will happen. Traditional UniswapV2 does not
    // do that so be careful!
    function migrate(IERC20 token) external returns (IERC20);
}
```

***IStakingRewards.sol***

```
pragma solidity ^0.6.2;

interface IStakingRewards {
    function balanceOf(address account) external view returns (uint256);

    function earned(address account) external view returns (uint256);

    function exit() external;

    function getReward() external;

    function getRewardForDuration() external view returns (uint256);

    function lastTimeRewardApplicable() external view returns (uint256);

    function lastUpdateTime() external view returns (uint256);

    function notifyRewardAmount(uint256 reward) external;

    function periodFinish() external view returns (uint256);

    function rewardPerToken() external view returns (uint256);

    function rewardPerTokenStored() external view returns (uint256);

    function rewardRate() external view returns (uint256);

    function rewards(address) external view returns (uint256);

    function rewardsDistribution() external view returns (address);
```

knownsec

```
    function rewardsDuration() external view returns (uint256);

    function rewardsToken() external view returns (address);

    function stake(uint256 amount) external;

    function stakeWithPermit(
        uint256 amount,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external;

    function stakingToken() external view returns (address);

    function totalSupply() external view returns (uint256);

    function userRewardPerTokenPaid(address) external view returns (uint256);

    function withdraw(uint256 amount) external;
}
```

***IStrategy.sol***

```
pragma solidity ^0.6.2;

interface IStrategy {//knownsec//  Strategy Pool Interface
    function want() external view returns (address);

    function deposit() external;

    function withdraw(address) external;

    function withdraw(uint256) external;

    function withdrawAll() external returns (uint256);

    function balanceOf() external view returns (uint256);

    function freeWithdraw(uint256 _amount) external;

    function harvest() external;

    function getHarvestable() external view returns (uint256);
}
```

***IStrategyConverter.sol***

```
pragma solidity ^0.6.2;

interface IStrategyConverter {
    function convert(
        address _refundExcess, // address to send the excess amount when adding liquidity
        address _fromWant,
        address _toWant,
        uint256 _wantAmount
    ) external returns (uint256);
}
```

***ISwerveFi.sol***

```
pragma solidity ^0.6.2;

interface ISwerveFi {
    function get_virtual_price() external view returns (uint);

    function add_liquidity(
        uint256[4] calldata amounts,
        uint256 min_mint_amount
    ) external;

    function remove_liquidity_imbalance(
        uint256[4] calldata amounts,
        uint256 max_burn_amount
    ) external;

    function remove_liquidity(
        uint256 _amount,
        uint256[4] calldata amounts
    ) external;
```

Knownsec

```
    function exchange(
        int128 from, int128 to, uint256 _from_amount, uint256 _min_to_amount
    ) external;

    function calc_token_amount(
        uint256[4] calldata amounts,
        bool deposit
    ) external view returns (uint);

    function calc_withdraw_one_coin(
        uint256 _token_amount, int128 i) external view returns (uint256);

    function remove_liquidity_one_coin(uint256 _token_amount, int128 i,
        uint256 min_amount) external;

    function balances(int128) external view returns (uint256);
}
```

***IUniswapV2Factory.sol***

```
pragma solidity ^0.6.2;

interface IUniswapV2Factory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    function feeTo() external view returns (address);
    function feeToSetter() external view returns (address);

    function getPair(address tokenA, address tokenB) external view returns (address pair);
    function allPairs(uint) external view returns (address pair);
    function allPairsLength() external view returns (uint);

    function createPair(address tokenA, address tokenB) external returns (address pair);

    function setFeeTo(address) external;
    function setFeeToSetter(address) external;
}
```

***IVault.sol***

```
pragma solidity ^0.6.2;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface IVault is IERC20 {
    function balance() external view returns (uint256);

    function balanceOfToken() external view returns (uint256);

    function token() external view returns (address);

    function claimInsurance() external; // NOTE: Only yDelegatedVault implements this

    function getRatio() external view returns (uint256);

    function deposit(uint256) external;

    function withdraw(uint256) external;

    function earn() external;
}
```

***IYfvRewards.sol***

```
pragma solidity ^0.6.2;

interface IYfvRewards {
    function balanceOf(address tokenAddress, address account) external view returns (uint256);

    function lastTimeRewardApplicable() external view returns (uint256);

    function weiTotalSupply() external view returns (uint256);

    function rewardPerToken() external view returns (uint256);

    function weiBalanceOf(address account) external view returns (uint256);

    function earned(address account) external view returns (uint256);
```

Knownsec

```
    function stakingPower(address account) external view returns (uint256);
    function vUSDBalance(address account) external view returns (uint256);
    function vETHBalance(address account) external view returns (uint256);
    function claimVETHReward() external;
    function stake(address tokenAddress, uint256 amount, address referrer) external;
    function withdraw(address tokenAddress, uint256 amount) external;
    function exit() external;
    function getReward() external;
    function nextRewardMultiplier() external view returns (uint16);
    function notifyRewardAmount(uint256 reward) external;
}
```

***OneSplitAudit.sol***

```
pragma solidity ^0.6.2;

interface OneSplitAudit {
    function getExpectedReturn(
        address fromToken,
        address toToken,
        uint256 amount,
        uint256 parts,
        uint256 featureFlags
    )
    external
    view
    returns (uint256 returnAmount, uint256[] memory distribution);

    function swap(
        address fromToken,
        address toToken,
        uint256 amount,
        uint256 minReturn,
        uint256[] calldata distribution,
        uint256 featureFlags
    ) external payable;
}
```

***UniswapRouterV2.sol***

```
pragma solidity ^0.6.2;

interface UniswapRouterV2 {
    function swapExactTokensForTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external returns (uint256[] memory amounts);

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    )
    external
    returns (
        uint256 amountA,
        uint256 amountB,
        uint256 liquidity
    );

    function addLiquidityETH(
        address token,
        uint256 amountTokenDesired,
        uint256 amountTokenMin,
        uint256 amountETHMin,
```

Knownsec

```
        address to,
        uint256 deadline
    )
    external
    payable
    returns (
        uint256 amountToken,
        uint256 amountETH,
        uint256 liquidity
    );

    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint256 liquidity,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    ) external returns (uint256 amountA, uint256 amountB);

    function getAmountsOut(uint256 amountIn, address[] calldata path)
    external
    view
    returns (uint256[] memory amounts);

    function getAmountsIn(uint256 amountOut, address[] calldata path)
    external
    view
    returns (uint256[] memory amounts);

    function swapETHForExactTokens(
        uint256 amountOut,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external payable returns (uint256[] memory amounts);

    function swapExactETHForTokens(
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external payable returns (uint256[] memory amounts);
}
```

***USDT.sol***

```
pragma solidity ^0.6.0;

interface USDT {
    function approve(address guy, uint256 wad) external;

    function transfer(address _to, uint256 _value) external;
}
```

knownsec

# 6. Appendix B: Vulnerability risk rating criteria

| *Smart contract vulnerability rating criteria* | |
|---|---|
| **Vulnerability rating** | **Vulnerability rating description** |
| **High-risk vulnerabilities** | Vulnerabilities that can directly cause losses of token contracts or users' funds, such as: numerical overflow loopholes that can cause the value of token to return to zero, false charging loopholes that can cause losses of tokens in exchanges, or ETH or re-entry loopholes in contract accounts;<br>Vulnerabilities that can cause the loss of escrow rights of token contracts, such as access control defects of key functions, access control byPass of key functions caused by call injection, etc.<br>Vulnerabilities that cause token contracts to not work properly, such as the denial of service vulnerability caused by sending the ETH to a malicious address, or the denial of service vulnerability caused by running out of gas. |
| **Medium-Dangerous Vulnerability** | High-risk vulnerabilities that require a specific address to trigger, such as numerical overflow vulnerabilities that can only be triggered by the owner of a token contract; Access control defects of non-critical functions, logical design defects that cannot cause direct capital loss, etc. |
| **Low-risk vulnerabilities** | Vulnerabilities that are difficult to be triggered, vulnerabilities that have limited harm after being triggered, such as numerical overflow vulnerabilities that require a large number of ETH or tokens to be triggered, vulnerabilities that the attacker cannot directly profit after triggering numerical overflow, and transaction sequence dependence risks triggered by specifying high gas, etc. |

Knownsec

# 6. Appendix C: Introduction to vulnerability testing tools

## 6.1 Manticore

A Manticore is a symbolic execution tool for analyzing binary files and smart contracts. A Manticore consists of a symbolic Ethereum virtual machine (EVM), an EVM disassembler/assembler, and a convenient interface for automatic compilation and analysis of the Solarium body.It also incorporates Ethersplay, a Bit of Traits of Bits visual disassembler for EVM bytecode, for visual analysis. Like binaries, Manticore provides a simple command-line interface and a Python API for analyzing EVM bytecode.

## 6.2 Oyente

Oyente is a smart contract analysis tool that can be used to detect common bugs in smart contracts, such as reentrancy, transaction ordering dependencies, and so on.More conveniently, Oyente's design is modular, so this allows power users to implement and insert their own inspection logic to check the custom properties in their contracts.

## 6.3 securify. Sh

Securify verifies the security issues common to Ethereum's smart contracts, such as unpredictability of trades and lack of input verification, while fully automated and analyzing all possible execution paths, and Securify has a specific language for identifying vulnerabilities that enables the securities to focus on current security and other reliability issues at all times.

## 6.4 Echidna

Echidna is a Haskell library designed for fuzzy testing EVM code.

knownsec

## 6.5 MAIAN

MAIAN is an automated tool used to find holes in Ethereum's smart contracts. MAIAN processes the bytecode of the contract and tries to set up a series of transactions to find and confirm errors.

## 6.6 ethersplay

Ethersplay is an EVM disassembler that includes correlation analysis tools.

## 6.7 IDA - evm entry

Ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

## 6.8 want - ide

Remix is a browser-based compiler and IDE that allows users to build ethereum contracts and debug transactions using Solidity language.

## 6.9 KnownSec Penetration Tester kit

KnownSec penetration tester's toolkit, developed, collected and used by KnownSec penetration tester engineers, contains batch automated testing tools, self-developed tools, scripts or utilization tools, etc. dedicated to testers.

knownsec