SMART CONTRACT AUDIT REPORT

for

BT.FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China

Jan. 7, 2021

Document Properties

Client	BT.Finance
Title	Smart Contract Audit Report
Target	BT.Finance
Version	1.0
Author	Chiachih Wu
Auditors	Chiachih Wu, Xudong Shao
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Jan. 7, 2021	Chiachih Wu	Final Release
1.0-rc3	Jan. 6, 2021	Chiachih Wu	Release Candidate #3
1.0-rc2	Jan. 5, 2021	Chiachih Wu	Release Candidate #2
1.0-rc1	Jan. 4, 2021	Chiachih Wu	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the **BT.Finance** protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About BT.Finance

BT.Finance is a DeFi aggregator farming tokens from other DeFi yield aggregators and platforms. As the main component of BT.Finance, the BT Vault is divided into 3 pools for various risk tolerances: Stable Profits Pool, High Yield Pool, and Smart Hybrid Pool. The BT Vault v1 is to earn $PICKLE tokens from Pickle.finance and $FARM tokens from Harvest.finance.

The basic information of BT.Finance is as follows:

Table 1.1: Basic Information of BT.Finance

Item	Description
Issuer	BT.Finance
Website	https://bt.finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Jan. 7, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit:

- https://github.com/btdotfinance/bt-finance (6300bc1)

- https://github.com/btdotfinance/bt-finance (3e6dd39)
- https://github.com/btdotfinance/bt-finance (ef9de30)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the BT.Finance implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	2	■ ■
Informational	5	■ ■ ■ ■ ■
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerabilities, 2 low-severity vulnerabilities, 5 informational recommendations.

Table 2.1: Key Audit Findings of BT.Finance Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Reentrancy Risks in bVault	Time and State	Confirmed
PVE-002	Info.	Improved Ownership Transition in Timelock	Coding Practices	Fixed
PVE-003	Info.	Improved Precision in AutoStake	Numeric Errors	Confirmed
PVE-004	Info.	Improved Authentication Mechanism in Strategy Contracts	Coding Practices	Fixed
PVE-005	Low	Leftover Rewards Inside Strategy Contracts	Coding Practices	Fixed
PVE-006	Info.	Redundant Fallback Mechanism in Strategy Contracts	Coding Practices	Fixed
PVE-007	Medium	Missing Authentication for Critical Functions in Strategy Contracts	Authentication Errors	Fixed
PVE-008	Info.	Other Suggestions	Coding Practices	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Reentrancy Risks in bVault

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `bVault`
- Category: Time and State [7]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `bVault` as an example, the `deposit()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 361) starts before effecting the update on internal states (line 370), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching `re-entrancy` via the very same `deposit()` function. Specifically, if another `deposit()` call is embedded inside an ongoing `deposit()` call before the token transfer in line 361, (e.g., `token` is an ERC-777) both of them get the same `_before` amount but the latter would re-count the former deposit amount and mint equivalent b-tokens.

```
357     function deposit(uint _amount) public onlyRestrictContractCall {
```

```
        require(_amount > 0, "Cannot deposit 0");
        uint _pool = balance();
        uint _before = token.balanceOf(address(this));
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint _after = token.balanceOf(address(this));
        _amount = _after.sub(_before); // Additional check for deflationary tokens
        uint shares = 0;
        if (totalSupply() == 0) {
            shares = _amount;
        } else {
            shares = (_amount.mul(totalSupply())).div(_pool);
        }
        _mint(msg.sender, shares);
        userDepoistTime[msg.sender] = now;
        if (token.balanceOf(address(this))>earnLowerlimit){
          earn();
        }
    }
```

Listing 3.1: bVault::deposit()

In the meantime, we should mention that the `bVault`'s b-tokens implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`. However, the `token` may need to be validated before being supported.

Recommendation Based on the design of the `deposit()` function, it's not feasible to apply the `checks-effects-interactions` pattern. We suggest to use the `reentrancy guard` on `deposit()` and `withdraw()` functions.

Status As per discussion with the team, considering current supported `token` contracts are all standard erc20s, the team choose to leave it as is.

3.2 Improved Ownership Transition in Timelock

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Timelock`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

In BT.Finance, the `Timelock` contract is used to replace the privileged accounts in main contracts such as `bVault` and `Controller`. While reviewing the implementation, we notice that the ownership transition uses a 2-step design which allows the `pendingAdmin` to claim the ownership by signing a transaction. However, there's a flaw that the `admin` cannot reset the `pendingAdmin` when she makes a mistake in the previous `setPendingAdmin()` call, which breaks the 2-step ownership transition logic.

```
236     function setPendingAdmin(address pendingAdmin_) public {
237         // allows one time setting of admin for deployment purposes
238         if (admin_initialized) {
239             require(msg.sender == address(this), "Timelock::setPendingAdmin: Call must
                    come from Timelock.");
240         } else {
241             require(msg.sender == admin, "Timelock::setPendingAdmin: First call must
                    come from admin.");
242             admin_initialized = true;
243         }
244         pendingAdmin = pendingAdmin_;

246         emit NewPendingAdmin(pendingAdmin);
247     }
```

Listing 3.2: Timelock::setPendingAdmin()

Specifically, the `admin_initialized` flag reflects the state of setting the `pendingAdmin` in line 241. Later on, the `pendingAdmin` could use the `acceptAdmin()` function to claim the ownership. When `admin_initialized` is set, only the `Timelock` contract itself could perform `setPendingAdmin()` (line 238). In the case that the `admin` set a wrong `pendingAdmin` (e.g., due to a typo), the `admin` needs to `queueTransaction()` and reset it with 12hrs or 24hrs delay, which is not feasible in the scenario of setting the `admin` to a multi-sig contract right after the `Timelock` contract is created.

Recommendation Set the `admin_initialized` flag in `acceptAdmin()`.

Status This issue has been addressed in this commit: 22d9d1e.

3.3 Improved Precision in AutoStake

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AutoStake`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-197 [1]

Description

In BT.Finance, the `AutoStake` contract allows users to `stake()` `lpToken` into the `rewardPool` and get earned tokens back with `exit()`. In particular, the `earned()` function allows the caller to check the amount of earned tokens. While reviewing the implementation, we notice that the calculation could be further improved to provide a more accurate result.

```
  function earned(address account) external view returns (uint256) {
    if (totalShares == 0) {
```

```
573         return 0;
574       }
575       uint256 totalBalance = rewardPool.balanceOf(address(this));
576       uint256 totalEarn = rewardPool.earned(address(this));
577       uint256 perShare = totalBalance.add(totalEarn).mul(unit).div(totalShares);

579       return perShare.mul(share[account]).div(unit);
580     }
```

Listing 3.3: AutoStake::earned()

Specifically, the `perShare` is computed as $(\frac{totalBalance+totalEarn}{totalShares}) \cdot unit$ in line 577. Later on, $(\frac{perShare \cdot share[account]}{unit})$ is returned in line 579, which is a multiplication on the result of a division, leading to precision loss. In fact, the `mul(unit)` and `div(unit)` could be removed with improved precision and reduced gas consumption.

Recommendation Calculate the earned share without `unit` as follows:

```
  function earned(address account) external view returns (uint256) {
    if (totalShares == 0) {
      return 0;
    }
    uint256 totalBalance = rewardPool.balanceOf(address(this));
    uint256 totalEarn = rewardPool.earned(address(this));
    return totalBalance.add(totalEarn).mul(share[account]).div(totalShares);
  }
```

Listing 3.4: AutoStake::earned()

Status As per discussion with the team, considering the current implementation is more readable and easier for maintenance, they choose to leave it as is.

3.4 Improved Authentication Mechanism in Strategy Contracts

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Strategy Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

In strategy contracts, the `withdraw()` function allows the privileged `Controller` to get `token` assets into the `vault`. While reviewing the implementation, we notice that the authentication mechanism is implemented in the underlying `_withdraw()` function (line 258 in the code below), which is not a common practice.

```
    function withdraw(uint _amount) external
    {
        uint amount = _withdraw(_amount);
        if (amount > _amount)
        {
            amount = _amount;
        }
        address _vault = Controller(controller).vaults(address(want));
        require(_vault != address(0), "!vault");
        IERC20(want).safeTransfer(_vault, amount);
    }
```

Listing 3.5: StrategyDAIUNIFarm::withdraw()

```
    function _withdraw(uint _amount) internal returns(uint) {
        require(msg.sender == controller, "!controller");
        uint amount = IERC20(want).balanceOf(address(this));
        if (amount < _amount) {
            _withdrawSome(_amount.sub(amount));
            amount = IERC20(want).balanceOf(address(this));
        }
        return amount;
    }
```

Listing 3.6: StrategyDAIUNIFarm::_withdraw()

In Solidity, we typically use modifiers (e.g., `onlyController`) for authentication checks on entry points (i.e., external/public functions). Internal functions such as `_withdraw()` are used as libraries without the check in most cases. Same logic applies to the `withdrawAll()` function.

Recommendation Use modifiers on external functions for authenticating the caller.

Status This issue has been addressed by using the `onlyController` modifier on external functions in this commit: ef9de30.

3.5 Leftover Rewards Inside Strategy Contracts

- ID: PVE-005
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: Strategy Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

In strategy contracts, the `redelivery()` function helps to exchange farming rewards into WETH and re-deposit them for further farming. Besides, part of the farming rewards are converted into BT tokens and sent to `Controller.rewards` address. While reviewing the implementation, we notice that the calculation leads to dust rewards left inside the strategy contracts.

```
320     function redelivery() internal {
321         uint256 reward = IERC20(Farm).balanceOf(address(this));
322         if (reward > redeliverynum) {
323             uint256 _2weth = reward.mul(80).div(100); //80%
324             uint256 _2bt = reward.mul(20).div(100);  //20%
325             _swapUniswap(Farm,weth, _2weth);
326             _redelivery();
327             _swapUniswap(Farm, bt,_2bt);
328             IERC20(bt).safeTransfer(Controller(controller).rewards(), IERC20(bt).
                    balanceOf(address(this)));
329         }
330         deposit();
331     }
```

Listing 3.7: StrategyDAIUNIFarm::redelivery()

Specifically, the `_2weth` and `_2bt` computed in line 323 and line 324 may not consume all the `reward` due to precision issues. One better solution is to compute one of them (e.g., `_2weth`) and derive the other one (e.g., `_2bt`) by subtracting the first one from `reward`, which also reduces gas consumption.

Recommendation Consume all the rest rewards in `redelivery()`.

```
    function redelivery() internal {
        uint256 reward = IERC20(Farm).balanceOf(address(this));
        if (reward > redeliverynum) {
            uint256 _2weth = reward.mul(80).div(100); //80%
            uint256 _2bt = reward.sub(_2weth);  //20%
            _swapUniswap(Farm,weth, _2weth);
            _redelivery();
            _swapUniswap(Farm, bt,_2bt);
            IERC20(bt).safeTransfer(Controller(controller).rewards(), IERC20(bt).
                balanceOf(address(this)));
        }
        deposit();
    }
```

Listing 3.8: StrategyDAIUNIFarm::redelivery()

Status This issue has been addressed by deriving `_2bt` from `(reward - _2weth)` in this commit: ef9de30.

3.6 Redundant Fallback Mechanism in Strategy Contracts

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Strategy Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

As mentioned in Section 3.4, the underlying `_withdraw()` function handles the real things of withdrawal for `withdraw()`.

```
function withdraw(uint _amount) external
{
    uint amount = _withdraw(_amount);
    if (amount > _amount)
    {
        amount = _amount;
    }
    address _vault = Controller(controller).vaults(address(want));
    require(_vault != address(0), "!vault");
    IERC20(want).safeTransfer(_vault, amount);
}
```

Listing 3.9: StrategyDAIUNIFarm::withdraw()

Inside `_withdraw()`, we notice that the case `amount < _amount` is handled by withdrawing the offset amount (i.e., `_amount - amount` of tokens via the `_withdrawSome()` function. However, the `withdraw()` function also checks `amount > _amount` and set `amount` to `_amount` as a fallback mechanism.

```
function _withdraw(uint _amount) internal returns(uint) {
    require(msg.sender == controller, "!controller");
    uint amount = IERC20(want).balanceOf(address(this));
    if (amount < _amount) {
        _withdrawSome(_amount.sub(amount));
        amount = IERC20(want).balanceOf(address(this));
    }
    return amount;
}
```

Listing 3.10: StrategyDAIUNIFarm::_withdraw()

We believe this mechanism could be further improved by taking care both `amount < _amount` and `amount >= _amount` cases inside `_withdraw()` such that the upper layer `withdraw()` function could use the return value directly without extra checks and fallback mechanism.

Recommendation Return the exact amount for withdrawal in `_withdraw()` and remove the fallback logic in `withdraw()`.

```
    function withdraw(uint _amount) external
    {
        uint amount = _withdraw(_amount);
        address _vault = Controller(controller).vaults(address(want));
        require(_vault != address(0), "!vault");
        IERC20(want).safeTransfer(_vault, amount);
    }
```

Listing 3.11: StrategyDAIUNIFarm::withdraw()

```
    function _withdraw(uint _amount) internal returns(uint) {
        require(msg.sender == controller, "!controller");
        uint amount = IERC20(want).balanceOf(address(this));
        if (amount < _amount) {
            _withdrawSome(_amount.sub(amount));
            amount = IERC20(want).balanceOf(address(this));
            return amount;
        }
        return _amount;
    }
```

Listing 3.12: StrategyDAIUNIFarm::_withdraw()

Status This issue has been addressed in this commit: fa97d3d.

3.7 Missing Authentication for Critical Functions in Strategy Contracts

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `StrategyPickleUSDC`,`StrategyPickleWBTC`
- Category: Authentication Errors [5]
- CWE subcategory: CWE-306 [2]

Description

In the `StrategyPickleUSDC` contract, the `Controller` is allowed to `withdraw()` arbitrary amount of USDC from Curve. In particular, after withdrawing enough 3Crv tokens, the `withdrawUnderlying()` function is called to `remove_liquidity()` from the 3Crv pool and exchange all assets into USDC.

```
    function withdrawUnderlying(uint256 _amount) public returns (uint) {
        IERC20(crvPla).safeApprove(curvefi, 0);
        IERC20(crvPla).safeApprove(curvefi, _amount);
        uint _before = IERC20(want).balanceOf(address(this));
        ICurveFi(curvefi).remove_liquidity(_amount, [0,uint256(0),0]);
```

```
        uint256 _ydai = IERC20(ydai).balanceOf(address(this));
        uint256 _yusdt = IERC20(yusdt).balanceOf(address(this));

        if(_ydai>0)
        {
            IERC20(ydai).safeApprove(curvefi, 0);
            IERC20(ydai).safeApprove(curvefi, _ydai);
            ICurveFi(curvefi).exchange(0, 1, _ydai, 0);
        }
        if(_yusdt>0)
        {
            IERC20(yusdt).safeApprove(curvefi, 0);
            IERC20(yusdt).safeApprove(curvefi, _yusdt);
            ICurveFi(curvefi).exchange(2, 1, _yusdt, 0);
        }

        uint _after = IERC20(want).balanceOf(address(this));

        return _after.sub(_before);
    }
```

Listing 3.13: StrategyPickleUSDC::withdrawUnderlying()

However, this crucial `withdrawUnderlying()` function is defined as a public function, which allows bad actors to impersonate the privileged `Controller` to convert 3Crv into USDC whenever the strategy contract has 3Crv balance. The same issue is also applicable to the `StrategyPickleWBTC` contract.

Recommendation Make `withdrawUnderlying()` an `internal` function or authenticate the caller.

Status This issue has been addressed by making `withdrawUnderlying()` an internal function in this commit: ef9de305.

3.8 Other Suggestions

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Strategy Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

In this section, we elaborate some other suggestions. First, we notice that each strategy contract has a `doApprove()` function which approves Uniswap to spend unlimited tokens withheld by the strategy contract. By doing that in the `constructor()`, the strategy contract could exchange tokens (e.g., PICKLE) to other tokens (e.g., USDC) while re-delivering tokens into existing DeFi platforms (e.g., Curve). However, since an unlimited spending is approved in the `constructor()`, there's no use case for invoking `doApprove()` from outside such that we don't need a `public doApprove()` function here.

```
    constructor() public {
        governance = tx.origin;
        controller = 0x03D2079c54967f463Fd6e89E76012F74EBC62615;
        doApprove();
        swap2BTRouting = [pickletoken,weth,bt];
        swap2TokenRouting = [pickletoken,weth,want];
    }

    function doApprove () public{
        IERC20(pickletoken).approve(unirouter, 0);
        IERC20(pickletoken).approve(unirouter, uint(-1));
    }
```

Listing 3.14: StrategyPickleUSDC.sol

Secondly, the `withdraw(address)` function is not implemented in all strategy contracts but documented as shown in the code snippet below:

```
/*

 A strategy must implement the following calls;

 - deposit()
 - withdraw(address) must exclude any tokens used in the yield - Controller role -
     withdraw should return to Controller
 - withdraw(uint) - Controller | Vault role - withdraw should always return to vault
 - withdrawAll() - Controller | Vault role - withdraw should always return to vault
 - balanceOf()

 Where possible, strategies must remain as immutable as possible, instead of updating
     variables, we update the contract by linking it in the controller

*/
```

Listing 3.15: StrategyPickleUSDC.sol

This also leaves an unused interface in the `interface Strategy` declarations in the `Controller` contract.

```
interface Strategy {
    function want() external view returns (address);
    function deposit() external;
    function withadraw(address) external;
    function withdraw(uint) external;
    function withdrawAll() external returns (uint);
    function balanceOf() external view returns (uint);
}
```

f

Recommendation Make `doApprove()` an `internal` function. And remove the unused interface(s) and obsolete comments.

Status This issue has been addressed by removing making `doApprove()` an `internal` function and removing the unused interface in this commit: ef9de30.

4 Conclusion

In this audit, we have analyzed the design and implementation of BT.Finance, a DeFi aggregator which helps users to farm tokens from existing DeFi platforms. The system presents a clean and consistent design that makes it a distinctive and valuable addition of innovation to current DeFi ecosystem. During the audit, all identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197.html.

[2] MITRE. CWE-306: Missing Authentication for Critical Function. https://cwe.mitre.org/data/definitions/306.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE CATEGORY: Authentication Errors. https://cwe.mitre.org/data/definitions/1211.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.