SMART CONTRACT AUDIT REPORT

for

XBE Vaults

Prepared By: Yiqun Chen

PeckShield
November 14, 2021

Document Properties

Client	XBE Finance
Title	Smart Contract Audit Report
Target	XBE Vaults
Version	1.0
Author	Xuxian Jiang
Auditors	Patrick Liu, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 14, 2021	Xuxian Jiang	Final Release
1.0-rc1	November 7, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the XBE Vaults 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 XBE

XBE aims to integrate the functions of Financial Securities, FIAT services, and DeFi in a complementary protocol that brings the best of the three worlds together to create greater value for each while allowing users from their respective worlds to maintain the level of risk and privacy that they are used to and prefer. The audited XBE Vaults acts as the central part of the incentive structure of the XBE ecosystem. It allows protocol users to invest while the protocol keeps track of an ever-growing pool with additional gains returned back to users. The gains are harvested by employing various strategies that are designed to automate the best yield farming opportunities available.

The basic information of the XBE Vaults protocol is as follows:

Table 1.1: Basic Information of The XBE Vaults Protocol

Item	Description
Name	XBE Finance
Website	https://xbe.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 14, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/XBEfinance/vaults.git (7954767)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/XBEfinance/vaults.git (TBD)

1.2 About PeckShield

PeckShield Inc. [13] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of 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 OWASP Risk Rating Methodology [12]:

- 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 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) [11], 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) or blockchain software, 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 1.3: 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

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 Logics	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 `XBE Vaults` 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	1	■
High	1	■
Medium	4	■ ■ ■ ■
Low	3	■ ■ ■
Informational	0	
Total	9	

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 critical-severity vulnerability, 1 high-severity vulnerability, 4 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key XBE Vaults Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Sybil Attacks To Drain Vault Rewards	Business Logic	Fixed
PVE-002	Medium	Permission-less BaseVaultV2::earn()	Business Logic	Confirmed
PVE-003	Medium	Incorrect Logic of Controller::setStrategy()	Business Logic	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-005	Low	Suggested SafeMath For Overflow Prevention	Coding Practices	Fixed
PVE-006	High	Improper Logic in InstitutionalEURxbVault::depositUnwrapped()	Business Logic	Fixed
PVE-007	Low	Asset Consistency Check Between Vault And Strategy	Coding Practices	Fixed
PVE-008	Medium	Invalid Slippage Control in Treasury And CvxCrvStrategy	Time and State	Fixed
PVE-009	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Sybil Attacks To Drain Vault Rewards

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `BaseVault`, `BaseVaultV2`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `XBE Vaults` protocol, there are two base vault contracts, i.e., `BaseVault` and `BaseVaultV2`, which are inherited and extended to implement customized vaults. Protocol users can stake their assets into these vaults and receive pool tokens in return. It comes to our attention that these pool tokens are ERC20-compliant ones that can be freely transferred from one user to another.

However, these pool tokens represent the ownership share on the vaults and may be used to claim respective rewards. However, the current implementation does not have the logic in place to properly settle the rewards before making the token transfers. As a result, the current logic is vulnerable to so-called Sybil attacks to drain all rewards in current vaults.

For elaboration, let's assume at the very beginning there is a malicious actor named `Malice`, who owns 100 `LP` tokens. `Malice` has an accomplice named `Trudy` who currently has 0 balance of `LP`. This `Sybil` attack can be launched as follows:

```
    function _transfer(
        address sender,
        address recipient,
        uint256 amount
    ) internal virtual {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _balances[sender] = _balances[sender].sub(
            amount,
            "ERC20: transfer amount exceeds balance"
```

```
        );
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }
```

Listing 3.1: ERC20Vault::_transfer()

1. Malice initially claims his rewards and then transfers 100 LPs to Trudy (or M_1), who can now claim the rewards one more time!

2. M_1 claims the rewards and then transfers 100 LPs to M_2, who can also claim the rewards one more time.

3. We can repeat by transferring M_i's 100 LPs balance to M_{i+1} who can also claim the rewards. In other words, we can effectively drain all vault rewards with new accounts created and iterated!

Recommendation To mitigate, it is necessary to accompany every single transfer() and transferFrom() with necessary logic to settle rewards By doing so, we can effectively mitigate the above Sybil attacks.

Status The issue has been addressed by settling the rewards before making the actual transfers.

3.2 Permission-less earn() in BaseVault And BaseVaultV2

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: BaseVault, BaseVaultV2
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The XBE Vaults protocol is in essence a decentralized asset management protocol with the investment subsystem inspired from the yearn.finance framework, hence sharing similar architecture with vaults, controller, and strategies. While examining the vault implementation (inside the BaseVault contract), we notice a potential force investment risk that has been exploited in earlier hacks, e.g., yDAI [14] and BT.Finance [1]. To elaborate, we show blow the related BaseVault::earn() routine.

Specifically, new strategy contracts of XBE Vaults have been designed and implemented to invest VC assets (held in vaults), harvest growing yields, and return any gains, if any, to the investors. In order to have a smooth investment experience, the vault contract has a dedicated function, i.e., earn(), that can be invoked to kick off the investment.

```
    /// @notice Transfer tokens to controller, controller transfers it to strategy and
        earn (farm)
    function earn() external virtual override {
        uint256 _bal = stakingToken.balanceOf(address(this));
        stakingToken.safeTransfer(address(_controller), _bal);
        _controller.earn(address(stakingToken), _bal);
        for (uint256 i = 0; i < _validTokens.length(); i++) {
            _controller.claim(address(stakingToken), _validTokens.at(i));
        }
    }
```

Listing 3.2: `BaseVault::earn()`

It comes to our attention that the `earn()` function is not guarded or can be invoked by any one to initiate the investment. If the configured strategy blindly invests the deposited funds into an imbalanced `Curve` pool, the strategy will not result in a profitable investment. In fact, earlier incidents (yDAI and BT hacks [14, 1]) have prompted the need of a guarded call to the `earn()` function. For the very same reason, we argue for the guarded call to `earn()` to block potential flashloan-assisted attacks. One mitigation will be to only allow for `EOA`-based trustworthy keepers.

Recommendation Ensure the `earn()` can only be called via a trusted entity. And take extra care in ensuring the vault assets will not be blindly deposited into a faulty strategy (that is currently not making any profit).

Status This issue has been confirmed.

3.3 Incorrect Logic of Controller::setStrategy()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Controller`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As mentioned earlier in Section 3.2, the `XBE Vaults` protocol shares a common `yearn.finance`-based farming architecture with `vaults`, `controller`, and `strategies`. While examining the current `Controller` logic, we notice the `setStrategy()` function is flawed.

To elaborate, we show below the implementation of the `setStrategy()` function. As the name indicates, this function is proposed to set new mapping between the investment token and the associated investment strategy. If there is a current investment strategy, there is a need to withdraw all funds from it to the vault. However, our analysis on the current implementation shows that the full investment amount is not withdrawn! In fact, only the controller balance of the investment token is returned.

```
    function setStrategy(address _token, address _strategy)
        external
        override
        onlyOwnerOrStrategist
    {
        require(approvedStrategies[_token][_strategy], "!approved");
        address _current = strategies[_token];
        if (_current != address(0)) {
            uint256 amount = IERC20(IStrategy(_current).want()).balanceOf(
                address(this)
            );
            IStrategy(_current).withdraw(amount);
            emit WithdrawToVaultAll(_token);
        }
        strategies[_token] = _strategy;
    }
```

Listing 3.3: `Controller::setStrategy()`

Recommendation Revise the above `setStrategy()` logic by taking out all funds from the old strategy during replacement.

Status The issue has been fixed by this commit: `f7e5815`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.4: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `VaultWithAutoStake::_autoStakeForOrSendTo()` routine as an example. This routine is designed to trigger default handling. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice (line 28): the first one reduces the allowance to 0; and the second one sets the new allowance.

```
22    function _autoStakeForOrSendTo(
23        address _token,
24        uint256 _amount,
25        address _receiver
26    ) internal {
27        if (_token == tokenToAutostake) {
28            IERC20(_token).approve(votingStakingRewards, _amount);
29            IAutoStakeFor(votingStakingRewards).stakeFor(_receiver, _amount);
30        } else {
31            IERC20(_token).safeTransfer(_receiver, _amount);
32        }
33    }
```

Listing 3.5: `VaultWithAutoStake::_autoStakeForOrSendTo()`

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of approve()/transferFrom() as well, i.e., safeApprove()/safeTransferFrom(). We highlight that this issue is present in a number of contracts, including BaseVault, BaseVaultV2, VaultWithAutoStake, Controller, VotingStakingRewards, etc.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve()/transfer()/transferFrom().

Status The issue has been fixed by this commit: f7e5815.

3.5 Suggested SafeMath For Overflow Prevention

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: VaultWithFees
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [3]

Description

SafeMath is a Solidity math library especially designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. Our analysis shows that the current implementation can be improved with the use of SafeMath.

Specifically, we show below the setClaimFeePercentage() function from the VaultWithFees contract. This function is used to configure the current claim fee. However, it is possible that when the given input _percentage is added with the sumClaimFee variable, it may need to overflow the computation.

```
    function setClaimFeePercentage(uint256 _index, uint64 _percentage)
        external
        onlyOwner
    {
        require(_index < claimFee.length, "indexOutOfBound");
        sumClaimFee = sumClaimFee + _percentage - claimFee[_index].percentage;
        claimFee[_index].percentage = _percentage;
    }
```

Listing 3.6: VaultWithFees::setClaimFeePercentage()

Recommendation Revise the setClaimFeePercentage() logic to properly validate the given input to avoid unnecessary overflow computation.

Status The issue has been fixed by this commit: f7e5815.

3.6 Improper Logic in InstitutionalEURxbVault::depositUnwrapped()

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: High
- Target: `InstitutionalEURxbVault`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `XBE Vaults` protocol has a dedicated `InstitutionalEURxbVault` contract for institutional investors. This contract has public functions to accept the institutional investment. While reviewing a specific public function, i.e., `depositUnwrapped()`, we notice an issue that may charge the investor twice for the investment amount.

In particular, we show below this function's implementation. It has a rather straightforward logic in firstly transferring the investment funds from the investor, then calling the internal helper `_deposit ()`, and finally returning the pool share back to the investor. It comes to our attention that the internal helper `_deposit()` also contains the logic of transferring the same amount again from the investing user!

```
    function depositUnwrapped(uint256 _amount) public onlyInvestor {
        IERC20(tokenUnwrapped).safeTransferFrom(
            _msgSender(),
            address(this),
            _amount
        );
        uint256 shares = _deposit(
            address(this),
            _convert(tokenUnwrapped, address(stakingToken), _amount)
        );
        _transfer(address(this), _msgSender(), shares);
    }
```

Listing 3.7: `InstitutionalEURxbVault::depositUnwrapped()`

Recommendation Transfer the investment amount only one in the above `depositUnwrapped()` function.

Status The issue has been fixed by this commit: `f7e5815`.

3.7 Asset Consistency Check Between Vault And Strategy

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Controller`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [2]

Description

In the `XBE Vaults` protocol, there is a one-to-one mapping between a `vault` and its `strategy`. To properly link a `vault` with its `strategy`, it is natural for the two to operate on the same underlying asset. For example, the `SushiVault` allows for `USDT`-based deposits and withdraws. The associated `strategy`, i.e., a `SushiStrategy`-based instance, naturally has `USDT` as the underlying asset. If these two have different underlying assets, the link should not be successful.

If we examine the `setStrategy()` routine in the `controller` contract, this routine allows for dynamic binding of the `vault` with a new `strategy` (line 173). A successful binding needs to satisfy a number of requirements. One specific example is shown as follows: `require(IVault(vaults[_token]).token ()== Strategy(_strategy).want())`. Apparently, this requirement guarantees the consistency of the underlying asset between the `vault` and its associated `strategy`.

```
159     function setStrategy(address _token, address _strategy)
160         external
161         override
162         onlyOwnerOrStrategist
163     {
164         require(approvedStrategies[_token][_strategy], "!approved");
165         address _current = strategies[_token];
166         if (_current != address(0)) {
167             uint256 amount = IERC20(IStrategy(_current).want()).balanceOf(
168                 address(this)
169             );
170             IStrategy(_current).withdraw(amount);
171             emit WithdrawToVaultAll(_token);
172         }
173         strategies[_token] = _strategy;
174     }
```

Listing 3.8: `Controller::setStrategy()`

However, if we examine the `constructor()` of various `strategy` contracts, the requirement of having the same underlying asset is not enforced. A new `strategy` deployment with an ill-provided list of arguments with an unmatched underlying asset may cause unintended consequences, including possible asset loss. With that, we suggest to maintain an invariant by ensuring the consistency of the underlying asset when a new `strategy` is being deployed or linked.

Recommendation Ensure the consistency of the underlying asset between the `vault` and its associated `strategy`.

Status The issue has been fixed by this commit: `a3ac200`.

3.8 Invalid Slippage Control in Treasury And CvxCrvStrategy

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Treasury`, `CvxCrvStrategy`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [5]

Description

As part of the investment logic, there is a constant need in `XBE Vaults` to convert one token to another. The current protocol is designed to interact with various `UniswapV2/Curve` pools for token conversion. Our analysis shows that the conversion can be improved by specifying effective slippage control to avoid unnecessary loss.

```
    function convertToRewardsToken(address _tokenAddress, uint256 amount)
        public
        override
        authorizedOnly
    {
        require(_tokensToConvert.contains(_tokenAddress), "tokenIsNotAllowed");

        address[] memory path = new address[](3);
        path[0] = _tokenAddress;
        path[1] = uniswapRouter.WETH();
        path[2] = rewardsToken;

        uint256 amountOutMin = uniswapRouter.getAmountsOut(amount, path)[0];
        amountOutMin = amountOutMin.mul(slippageTolerance).div(MAX_BPS);

        IERC20 token = IERC20(_tokenAddress);
        if (token.allowance(address(this), address(uniswapRouter)) == 0) {
            token.approve(address(uniswapRouter), uint256(-1));
        }
        uniswapRouter.swapExactTokensForTokens(
            amount,
            amountOutMin,
            path,
            address(this),
            block.timestamp + swapDeadline
        );
        emit FundsConverted(_tokenAddress, rewardsToken, amountOutMin);
```

```
    }
```

Listing 3.9: `Treasury::convertToRewardsToken()`

To elaborate, we show above the `convertToRewardsToken()` routine from the `Treasury` contract. We notice the token swap is routed to `uniswapRouter` and the actual swap operation `swapExactTokensForTokens()` essentially specifies no restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller return. Note the way to use the `amountOutMin` parameter is invalid as it is computed from `getAmountsOut()`! In other words, the `getAmountsOut()` output guarantees `amountOut=amountOutMin`, regardless of the given `slippageTolerance`! Other functions share similar issue, including `CvxCrvStrategy::convertAndStakeTokens()`.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been fixed by this commit: `a3ac200`.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [4]

Description

In the `XBE Vaults` protocol, the privileged owner account plays a critical role in governing and regulating the system-wide operations (e.g., `vault/strategy` addition, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets for investment or full withdrawal among the three components, i.e., `vault`, `controller`, and `strategy`.

With great privilege comes great responsibility. Our analysis shows that the governance account is indeed privileged. In the following, we examine the current privilege management graph in the XBE Vaults protocol (Figure 3.1)[1].

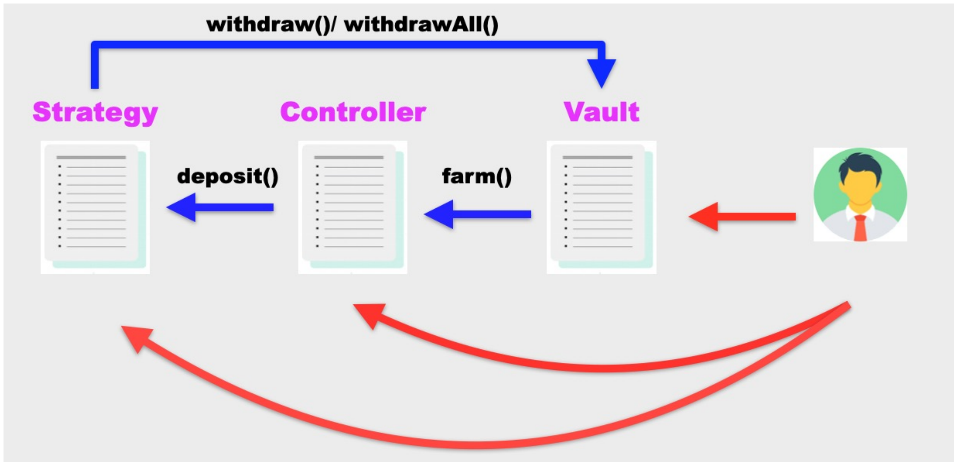

Figure 3.1: The Privilege Management Chain in XBE Vaults

We emphasize that the privilege assignment among vault, controller, and strategy is properly administrated. However, it is worrisome if the governance is not governed by a DAO-like structure. The discussion with the team has confirmed that the governance will be managed by a multi-sig account.

We point out that a compromised owner account would allow the attacker to add a malicious controller to steal all funds whenever the earn() call is made. It could also allow for the dynamic addition of a new malicious strategy, which directly undermines the assumption of the entire protocol.

Recommendation Promptly transfer the owner privilege to the intended DAO-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed and partially mitigated with a multi-sig account to regulate the governance/controller privileges.

[1]Note the farm() operation is actually implemented as earn().

4 Conclusion

In this audit, we have analyzed the design and implementation of the `XBE Vaults` protocol. The audited system presents a unique addition to current DeFi offerings by integrating the functions of `Financial Securities`, `FIAT services`, and `DeFi` into a complementary protocol. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity`-based 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] BT Finance. BT.Finance Exploit Analysis Report. https://btfinance.medium.com/bt-finance-exploit-analysis-report-a0843cb03b28.

[2] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[3] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[4] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

[9] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[10] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

[14] PeckShield. The yDAI Incident Analysis: Forced Investment. https://peckshield.medium.com/the-ydai-incident-analysis-forced-investment-2b8ac6058eb5.