SMART CONTRACT AUDIT REPORT

for

WHITEHEART

Prepared By: Shuxiao Wang

PeckShield

03/03/2021

Document Properties

Client	Hegic
Title	Smart Contract Audit Report
Target	Whiteheart
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Jeff Liu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	03/03/2021	Xuxian Jiang	Final Release
1.0-rc	03/02/2021	Xuxian Jiang	Release Candidate #1
0.2	03/01/2021	Xuxian Jiang	Additional Findings
0.1	02/25/2021	Xuxian Jiang	Initial Draft

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 **Whiteheart Protocol** design document and related smart contract source code, we in the report outline 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 Whiteheart Protocol

`Whiteheart` is an on-chain hedging protocol built on top of the `Hegic` protocol. The core part of the hedging protocol is a new financial primitive called `hedge contract` that can automatically conduct the process of hedging users' holdings' market value. `Hedge contracts` utilize liquidity which is pooled by liquidity providers on non-custodial smart contracts to act as the value downside insurance sellers and earn fees paid by hedge contract buyers in case the value of assets will not decrease.

The basic information of Whiteheart is as follows:

Table 1.1: Basic Information of Whiteheart

Item	Description
Issuer	Hegic
Website	https://www.whiteheart.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	03/03/2021

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

- https://github.com/jmonteer/whiteheart-v1.git (bf7759c)

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

- https://github.com/jmonteer/whiteheart-v1.git (d1a0187)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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.

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

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) [10], 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 1.4: 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 Whiteheart Protocol design and 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	2	■■
Low	4	■■■■
Informational	2	■■
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 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 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities and 2 informational recommendations.

Table 2.1: Key Whiteheart Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested SafeMath Usage	Coding Practices	Fixed
PVE-002	Low	Uninitialized autoUnwrapDisabled	Business Logic	Fixed
PVE-003	Low	No Slippage Control in _createHedge()	Time And State	Fixed
PVE-004	Informational	Improved Logic in _receiveAsset()	Business Logic	Fixed
PVE-005	Medium	Admin Key Trust on USDC Pool Owner	Security Features	Fixed
PVE-006	Medium	Lockup-Free WhiteStaking::withdraw()	Business Logic	Fixed
PVE-007	Low	Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()	Coding Practices	Fixed
PVE-008	Informational	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Suggested SafeMath Usage

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: WHAssetv2
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

SafeMath is a Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. While examining current math operations, we notice an occasion that can benefit from the use of SafeMath.

To elaborate, we show below the constructor() of the WHAssetv2 contract. The internal result of _DECIMALS is computed as 10 ** (IToken(_token).decimals()- IToken(_stablecoin).decimals())* PRICE_DECIMALS, which may overflow if the given _token has a smaller decimal than the given _stablecoin.

```
    constructor(
            IUniswapV2Router02 _swapRouter,
            IToken _stablecoin,
            IToken _token,
            AggregatorV3Interface _priceProvider,
            IWhiteUSDCPool _pool,
            IWhiteOptionsPricer _whiteOptionsPricer,
            string memory _name,
            string memory _symbol) public ERC721(_name, _symbol)
    {
        uint _DECIMALS = 10 ** (IToken(_token).decimals() - IToken(_stablecoin).decimals
            ()) * PRICE_DECIMALS;
        DECIMALS = _DECIMALS;
```

```
        address [] memory _underlyingToStableSwapPath = new address [](2);
        _underlyingToStableSwapPath [0] = address(_token);
        _underlyingToStableSwapPath [1] = address(_stablecoin);

        underlyingToStableSwapPath = _underlyingToStableSwapPath;

        swapRouter = _swapRouter;
        whiteOptionsPricer = _whiteOptionsPricer;
        priceProvider = _priceProvider;
        stablecoin = _stablecoin;
        pool = _pool;
    }
```

Listing 3.1: WHAssetv2::**constructor**()

Recommendation Revise the above logic by using the `SafeMath` library.

Status This issue has been fixed in the commit: 671283e.

3.2 Uninitialized autoUnwrapDisabled

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WHAssetv2`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The `Whiteheart` protocol provides helper routines to facilitate the wrapping of principal into a `hedge contract` as well as the reverse operation of unwrapping. The wrapped principal amount is therefore insured or protected against sudden price drop. While examining the unwrapping support, we notice the internal state that allows for users to disable the automatic unwrapping is uninitialized before the use.

To elaborate, we show below the affected `isAutoUnwrapable()` routine. As the name indicates, this routine aims to answer the question on whether the hedge contract is auto-unwrappable. And it is part of the current logic to query internal state of `autoUnwrapDisabled[_underlying.owner]` (line 205), which unfortunately is not initialized. In other words, the check at line 205 is simply a no-op.

```
198     /**
199      * @notice Answer the question: is this hedge contract Auto unwrappable?
200      * @param tokenId HedgeContract to be unwrapped
201      * @return answer to the question: is this hedge contract Auto unwrappable
202      */
```

```
    function isAutoUnwrapable(uint tokenId) public view returns (bool) {
        Underlying memory _underlying = underlying[tokenId];
        if(autoUnwrapDisabled[_underlying.owner]) return false;
        if(!_underlying.active) return false;

        bool ITM = false;
        uint currentPrice = _currentPrice();

        ITM = currentPrice < _underlying.strike;

        // if option is In The Money and the option is going to expire in the next
            minutes
        if (ITM && ((_underlying.expiration.sub(30 minutes) <= block.timestamp) && (
            _underlying.expiration >= block.timestamp))) {
            return true;
        }

        return false;
    }
```

Listing 3.2: WHAssetv2::isAutoUnwrapable()

Recommendation Add necessary `Setters` to allow for the initialization of `autoUnwrapDisabled`.

Status This issue has been fixed in the commit: 671283e.

3.3 No Slippage Control in _createHedge()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WHAssetv2`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [4]

Description

`Whiteheart` is an on-chain hedging protocol that automatically buys an `at-the-money` (`ATM`) `put` option contract. A `put` option is a right but not an obligation to sell an asset at a fixed price during a certain period of time. Hedging with `at-the-money` (`ATM`) `put` options means that the strike price of an option at the moment of protecting an asset's value will be equal to the market price of the asset.

To elaborate, we show below the `_createHedge()` routine that actually instantiates a hedge contract. We notice this routine internally makes use of the `swapRouter` to swap the token to the underlying stablecoin (lines 274 – 280).

```
261     function _createHedge(uint tokenId, uint totalFee, uint settlementFee, uint period,
            uint amount, uint strike, address owner) internal {
```

```
        uint collateral = amount.mul(strike).mul(optionCollateralizationRatio).div(100).div(DECIMALS);

        Underlying memory _newUnderlying = Underlying(
            bool(true),
            address(owner),
            uint88(amount),
            uint48(block.timestamp + period),
            uint48(strike)
        );
        underlying[tokenId] = _newUnderlying;

        uint premiumPercentage = totalFee.sub(settlementFee).mul(1000).div(totalFee);
        uint[] memory amounts = swapRouter.swapExactTokensForTokens(
            totalFee,
            0,
            underlyingToStableSwapPath,
            address(pool),
            block.timestamp
        );
        uint totalStablecoin = amounts[amounts.length - 1];
        uint premiumStablecoin = premiumPercentage.mul(totalStablecoin).div(1000);
        uint settlementFeeStablecoin = totalStablecoin.sub(premiumStablecoin);

        pool.lock(tokenId, collateral, premiumStablecoin, settlementFeeStablecoin);
    }
```

Listing 3.3: WHAssetv2::_createHedge()

We observe that there is no slippage control in place, which opens up the possibility for front-running and potentially results in a smaller converted amount. 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. 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 sandwich arbitrage to better protect the interests of users.

Status This issue has been fixed in the commit: 671283e.

3.4 Improved Logic in _receiveAsset()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: WHERC20v2
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.3, Whiteheart is an on-chain hedging protocol that automatically buys an at-the-money (ATM) put option contract. The option contract buyers will need to pay associated fee. Using the WHAssetv2 contract as an example, the _wrap() routine is given the principal amount and option period and the purpose here is to wrap the insured principal to a hedge contract.

```
    function _wrap(uint amount, uint period, address to, bool receiveAsset, bool
        _mintToken) internal returns (uint newTokenId){
        // new tokenId
        _tokenIds.increment();
        newTokenId = _tokenIds.current();

        // get cost of option
        uint strike = _currentPrice();

        (uint total, uint settlementFee, , ) = whiteOptionsPricer.getOptionPrice(period,
            amount, strike);

        // receive asset + cost of hedge
        if(receiveAsset) _receiveAsset(msg.sender, amount, total);
        // buy option
        _createHedge(newTokenId, total, settlementFee, period, amount, strike, to);

        // mint ERC721 token
        if(_mintToken) _mint(to, newTokenId);

        emit Wrap(to, uint32(newTokenId), uint88(total), uint88(amount), uint48(strike),
            uint32(block.timestamp+period));
    }
```

Listing 3.4: WHAssetv2::_wrap()

When evaluating the fund movement, we notice there is an internal helper routine _receiveAsset() that is designed to transfer the cost, i.e., principal+hedge, from the buyer to the contract itself. It comes to our attention that the given buyer information, i.e., the first argument from of the helper routine, is not used in the actual asset transfer (line 57). It is more natural to simply use the function argument from instead of restricting the source as msg.sender.

```
    /**
     * @notice internal function that supports the receival of principal+hedge cost to
       be sent
     * @param from address sender
     * @param amount principal to receive
     * @param toUsdc hedgeCost
     */
    function _receiveAsset(address from, uint amount, uint toUsdc) internal override {
        token.safeTransferFrom(msg.sender, address(this), amount.add(toUsdc));
    }
```

Listing 3.5: _receiveAsset()

Recommendation Revise the `_receiveAsset()` logic to use its own arguments, instead of `msg.sender`. An example revision is shown below:

```
    /**
     * @notice internal function that supports the receival of principal+hedge cost to
       be sent
     * @param from address sender
     * @param amount principal to receive
     * @param toUsdc hedgeCost
     */
    function _receiveAsset(address from, uint amount, uint toUsdc) internal override {
        token.safeTransferFrom(from, address(this), amount.add(toUsdc));
    }
```

Listing 3.6: _receiveAsset()

Status This issue has been fixed in the commit: 671283e.

3.5 Admin Key Trust on USDC Pool Owner

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `WHAssetv2`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `Whiteheart` protocol, there is a privileged account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter settings). Specifically, our analysis with the `USDC` pool shows that there is an `owner` account that can set up the lockup period as well as whitelist specific `whAsset` addresses to open positions using the USDC pool.

To elaborate, we show below the `_exercise()` routine that is responsible for exercising the `put` options. The exercise operation involves the unlocking of pool assets after option expiration (line 312) or the profit payment (line 315) before expiration.

```
307     function _exercise(uint tokenId, address owner) internal returns (uint optionProfit,
            uint amount) {
308         Underlying storage _underlying = underlying[tokenId];
309         amount = _underlying.amount;
310 
311         if(_underlying.expiration < block.timestamp){
312             pool.unlock(tokenId);
313             optionProfit = 0;
314         } else {
315             optionProfit = _payProfit(owner, tokenId, _underlying.strike, _underlying.
                    amount);
316         }
317     }
```

Listing 3.7: WHAssetv2::_exercise()

However, both unlocking (via `pool.unlock()`) and profit payment (via `pool.send()`) are guarded with a modifier, i.e,. `onlyWHAssets`. This modifier is regulated by the powerful `owner` account. In other words, the funds from option buys may be locked in the contract if the `owner` account somehow remove the `whAsset` addresses from being whitelisted to open a position.

```
    function unlock(uint256 id) external override onlyWHAssets {
        LockedLiquidity storage ll = lockedLiquidity[msg.sender][id];
        require(ll.locked, "LockedLiquidity with such id has already unlocked");
        ll.locked = false;
        lockedPremium = lockedPremium.sub(ll.premium);
        lockedAmount = lockedAmount.sub(ll.amount);
        emit Profit(id, ll.premium);
    }
```

Listing 3.8: WhiteUSDCPool::unlock()

```
    function send(uint id, address payable to, uint256 amount, uint _payKeep3r)
    external override onlyWHAssets
    {
        LockedLiquidity storage ll = lockedLiquidity[msg.sender][id];
        require(ll.locked, "LockedLiquidity with such id has already unlocked");
        require(to != address(0));

        ll.locked = false;
        lockedPremium = lockedPremium.sub(ll.premium);
        lockedAmount = lockedAmount.sub(ll.amount);

        uint transferAmount = amount > ll.amount ? ll.amount : amount;
        token.safeTransfer(to, transferAmount.sub(_payKeep3r));

        if(_payKeep3r > 0) owedToKeep3r = owedToKeep3r.add(_payKeep3r);
```

```

        if (transferAmount <= ll.premium)
            emit Profit(id, ll.premium - transferAmount);
        else
            emit Loss(id, transferAmount - ll.premium);
    }
```

Listing 3.9: WhiteUSDCPool::send()

Recommendation While it is appropriate to have a whitelist capability to open a position, the close operation should not be blocked. In other words, the above two functions, i.e., `unlock()` and `send()`, do not need the `onlyWHAssets` modifier.

Status This issue has been fixed by removing the `onlyWHAssets` modifier from `unlock()` and `send()` functions.

3.6 Lockup-Free WhiteStaking::withdraw()

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `WhiteStaking`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

By design, the Whiteheart protocol will generate and collect settlement fees paid every time when an `ATM put` option contract is purchased. To encourage the protocol adoption, the protocol has a built-in staking-based incentivizer mechanism as demonstrated in the `WhiteStakingUSDC` contract.

In order to prevent possible flashloan-assisted sandwich-style arbitrages that may claim the majority of rewards, the staking logic is designed to have a lockup period for staked assets. For each account, the associated lockup period is recorded as `[lastBoughtTimestamp[account], lastBoughtTimestamp[account].add(lockupPeriod)]`.

```
    function deposit(uint amount) external override {
       lastBoughtTimestamp[msg.sender] = block.timestamp;
        require(amount > 0, "!amount");
        WHITE.safeTransferFrom(msg.sender, address(this), amount);

        _mint(msg.sender, amount);
    }

    function withdraw(uint amount) external override {
        _burn(msg.sender, amount);
```

```
        WHITE.safeTransfer(msg.sender, amount);
    }
```

Listing 3.10: WhiteStaking::deposit() and WhiteStaking::withdraw()

However, it comes to our attention that the unstaking function, i.e., `withdraw()`, does not honor the lockup period, which completely defeat the purpose of the lockup period design. To mitigate, we suggest to use the `lockupFree` modifier with the `withdraw()` routine.

Recommendation Properly enforce the `lastBoughtTimestamp` when a staking user attempts to withdraw the staked assets.

Status This issue has been fixed in the commit: 671283e.

3.7 Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WHAssetv2`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

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 `transfer()` 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.

```
/**
* @dev transfer token for a specified address
* @param _to The address to transfer to.
* @param _value The amount to be transferred.
*/
function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
    uint fee = (_value.mul(basisPointsRate)).div(10000);
    if (fee > maximumFee) {
        fee = maximumFee;
    }
    uint sendAmount = _value.sub(fee);
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
```

```
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.11: USDT Token **Contract**

It is important to note the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer(address recipient, uint256 amount)external returns (bool)`. 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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `provide()` routine in the `WhiteUSDCPool` contract. If `USDT` is given as `token`, the unsafe version of `token.transferFrom(msg.sender, address(this), amount)` (line 171) may revert as there is no return value in the `USDT` token contract's `transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
150    /**
151     * @notice A provider supplies USDC to the pool and receives writeUSDC tokens
152     * @param amount Amount to send to the contract
153     * @param minMint minimum amount of writeUSDC tokens to be minted
154     * @return mint amount of writeUSDC minted to provider
155     */
156    function provide(uint256 amount, uint256 minMint) external returns (uint256 mint) {
157        lastProvideTimestamp[msg.sender] = block.timestamp;
158        uint supply = totalSupply();
159        uint balance = totalBalance();
160        if (supply > 0 && balance > 0)
161            mint = amount.mul(supply).div(balance);
162        else
163            mint = amount.mul(INITIAL_RATE);

165        require(mint >= minMint, "Pool: Mint limit is too large");
166        require(mint > 0, "Pool: Amount is too small");
167        _mint(msg.sender, mint);
168        emit Provide(msg.sender, amount, mint);

170        require(
171            token.transferFrom(msg.sender, address(this), amount),
```

```
            "Token transfer error: Please lower the amount of premiums that you want to
                send."
        );
    }
```

Listing 3.12: WhiteUSDCPool::provide()

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

Status This issue has been fixed in the commit: 671283e.

3.8 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `WhiteStaking`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

In the `Whiteheart` protocol, the `WhiteStaking` contract is designed to be the main entry for interaction with staking users. In particular, one entry routine, i.e., `deposit()`, accepts user deposits of supported assets (e.g., `DAI`). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `WhiteStaking` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function deposit(uint amount) external override {
       lastBoughtTimestamp[msg.sender] = block.timestamp;
        require(amount > 0, "!amount");
        WHITE.safeTransferFrom(msg.sender, address(this), amount);

        _mint(msg.sender, amount);
    }

    function withdraw(uint amount) external override {
        _burn(msg.sender, amount);

        WHITE.safeTransfer(msg.sender, amount);
    }
```

Listing 3.13: WhiteStaking::deposit() and WhiteStaking::withdraw()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into the `WhiteStaking`. In our case, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()`/`transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status This issue has been confirmed. However, considering the fact that this specific issue does not affect the normal operation, the team decides to address it when the need of supporting deflationary/rebasing tokens arises.

4 Conclusion

In this audit, we have analyzed the Whiteheart design and implementation. The system presents a unique offering in current DeFi ecosystem in automatically conducting the process of hedging users' holdings' market value. The current code base is well organized and those 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

[4] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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