SMART CONTRACT AUDIT REPORT

for

Tortle Ninja

Prepared By: Xiaomi Huang

PeckShield

August 01, 2022

Document Properties

Client	Tortle Ninja
Title	Smart Contract Audit Report
Target	Tortle Ninja
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 01, 2022	Xuxian Jiang	Final Release
1.0-rc	June 20, 2022	Luck Hu	Release Candidate

Contact

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

Name	Xiaomi Huang
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 Tortle Ninja protocol, we outline in this 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 contract can be further improved due to the presence of the identified issues. This document outlines our audit results.

1.1 About Tortle Ninja

Tortle Ninja is a no-code DeFi meta aggregator which allows the creation of Tortle Combo Recipes, which is a combination of DeFi operations that enable you to create, shoot, repeat, and copy complex investment strategies. And Tortle runs natively on Fantom Opera which is by far the most reliable, open, and fast network. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Tortle Ninja

Item	Description
Name	Tortle Ninja
Website	https://tortle.ninja//
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 01, 2022

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

- https://github.com/Tortle-Ninja/Tortle-Contracts.git (2a83ab2)

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

- https://github.com/Tortle-Ninja/Tortle-Contracts.git (ece2d86)

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.

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

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

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 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 implementation of the `Tortle Ninja` protocol. 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	3	■ ■ ■
Informational	0	
Total	5	

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 and 3 low-severity vulnerabilities.

Table 2.1: Key Tortle Ninja Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Validation of Function Arguments	Business Logic	Fixed
PVE-002	Low	Missed Access Control in Nodes_::split()	Coding Practices	Confirmed
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Lack of Slippage Control in Tortle	Time and State	Fixed

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 Improved Validation of Function Arguments

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Nodes_
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

In the Nodes contract, we observe the depositOnLp() routine is used to deposit user assets (token0, token1) to UniswapV2 to add liquidity. To elaborate, we show below the related code snippet of the depositOnLp(). It calls the _addLiquidity() routine (line 107) to add liquidity to UniswapV2, and the received LP tokens will be credited to user by updating the userLp[lpToken][user] (line 108) which represents the total amount of lpToken deposited for user. However, we notice the lack of proper validation for the parameters: lpToken, token0 and token1 which may lead to user assets loss.

Specifically, according to the design, the lpToken shall be the same pair of (router.factory, token0, token1). Without this validation, the routine caller can add liquidity to UniswapV2 with two arbitrary tokens, but receive any amount of the given lpToken (usually much valuable) as the received LP tokens. After that, user could then withdraw all the valuable assets in this contract with arbitrary tokens. Currently the routine could only be called from owner or batch, which greatly alleviate this concern. However, we still recommend to add the above mentioned parameters validation.

```
97      function depositOnLp(
98          address user,
99          address lpToken,
100         address token0,
101         address token1,
102         uint256 amount0,
103         uint256 amount1
104     ) external nonReentrant onlyOwner returns (uint256) {
```

```
        require(amount0 <= getBalance(user, IERC20(token0)), 'DepositOnLp: Insufficient token0 funds.');
        require(amount1 <= getBalance(user, IERC20(token1)), 'DepositOnLp: Insufficient token1 funds.');
        (uint256 amount0f, uint256 amount1f, uint256 lpRes) = _addLiquidity(token0, token1, amount0, amount1);
        userLp[lpToken][user] += lpRes;
        decreaseBalance(user, address(token0), amount0f);
        decreaseBalance(user, address(token1), amount1f);
        return lpRes;
    }
```

Listing 3.1: `Nodes::depositOnLp()`

Similarly, it shares the same issue in routine `_withdrawLpAndSwap()`. And it could be improved by adding proper parameters validation to ensure that the `args.token0` and `args.token1` are both the underlying tokens of `args.lpToken`. Moreover, it's also suggested to add proper validation to ensure the `args.tokenDesired` is one of `args.token0 or args.token1`.

```
    function _withdrawLpAndSwap(
        address user,
        _wffot memory args,
        uint256 amountLp
    ) internal returns (uint256 amountTokenDesired) {
        IERC20(args.lpToken).transfer(args.lpToken, amountLp);
        (uint256 amount0, uint256 amount1) = IUniswapV2Pair(args.lpToken).burn(address(this));

        require(amount0 >= minimumAmount, 'UniswapV2Router: INSUFFICIENT_A_AMOUNT');
        require(amount1 >= minimumAmount, 'UniswapV2Router: INSUFFICIENT_B_AMOUNT');

        uint256 swapAmount;
        address swapToken;

        if (args.token1 == args.tokenDesired) {
            swapToken = args.token0;
            swapAmount = amount0;
            amountTokenDesired += amount1;
        } else {
            swapToken = args.token1;
            swapAmount = amount1;
            amountTokenDesired += amount0;
        }

        address[] memory path = new address[](2);
        path[0] = swapToken;
        path[1] = args.tokenDesired;

        _approve(swapToken, address(router), swapAmount);

        uint256[] memory swapedAmounts = router.swapExactTokensForTokens(
            swapAmount,
```

```
            args.amountTokenDesiredMin,
            path,
            address(this),
            block.timestamp
        );
        amountTokenDesired += swapedAmounts[1];
        increaseBalance(user, args.tokenDesired, amountTokenDesired);
    }
```

Listing 3.2: `Nodes::_withdrawLpAndSwap()`

Recommendation Add the above mentioned parameters validations at the beginning of the functions.

Status The issue has been fixed by the following commits: `b287bb8` and `cdc9254`.

3.2 Missed Access Control in Nodes_::split()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Nodes`
- Category: Coding Practices [8]
- CWE subcategory: CWE-628 [4]

Description

The `Nodes_` contract provides a routine (i.e. `split()`) to divide one token into two tokens according to the selected percentage. Before calling this routine, the user has to send the token to this contract. However there's a lack of proper access control validation at the beginning of this routine, which results in potential assets loss in this contract if user doesn't transfer in enough tokens first. Although currently this contract is not designed to store user assets, which greatly alleviates this concern, it's still suggested to add proper access control to the `split()` routine. Per our further study in the protocol, we can only allow the `Nodes` contract to call this routine.

```
    function split(
        address _token,
        uint256 _amount,
        address _firstToken,
        address _secondToken,
        uint256 _percentageFirstToken,
        uint256 _amountOutMinFirst,
        uint256 _amountOutMinSecond
    )
        public
        nonReentrant
        returns (uint256 amountOutToken1, uint256 amountOutToken2)
```

```
    {
        uint256 _firstTokenAmount = mulScale(
            _amount,
            _percentageFirstToken,
            10000
        ); // Amount of first token.
        uint256 _secondTokenAmount = _amount - _firstTokenAmount; // Amount of second token.

        IERC20(_token).approve(address(router), _amount);

        uint256[] memory amountsOut;
        uint256 _amountOutToken1;
        uint256 _amountOutToken2;
        if (_token == FTM) {
            (_amountOutToken1, _amountOutToken2) = _splitFromFTM(
                _firstToken,
                _secondToken,
                _firstTokenAmount,
                _secondTokenAmount,
                _amountOutMinFirst,
                _amountOutMinSecond
            );
        } else if (_firstToken == FTM   _secondToken == FTM) {
            (_amountOutToken1, _amountOutToken2) = _splitToFTM(
                _token,
                _firstToken,
                _secondToken,
                _firstTokenAmount,
                _secondTokenAmount,
                _amountOutMinFirst,
                _amountOutMinSecond
            );
        } else {
            if (_firstToken != _token) {
                address[] memory pathFirstToken = new address[](3);
                pathFirstToken[0] = _token;
                pathFirstToken[1] = FTM;
                pathFirstToken[2] = _firstToken;

                amountsOut = router.swapExactTokensForTokens(
                    _firstTokenAmount,
                    _amountOutMinFirst,
                    pathFirstToken,
                    address(msg.sender),
                    block.timestamp
                );

                _amountOutToken1 = amountsOut[amountsOut.length - 1];
            } else {
                _amountOutToken1 = _firstTokenAmount;
                IERC20(_firstToken).transfer(msg.sender, _firstTokenAmount);
```

```
            }

            if (_secondToken != _token) {
                address[] memory pathSecondToken = new address[](3);
                pathSecondToken[0] = _token;
                pathSecondToken[1] = FTM;
                pathSecondToken[2] = _secondToken;

                amountsOut = router.swapExactTokensForTokens(
                    _secondTokenAmount,
                    _amountOutMinSecond,
                    pathSecondToken,
                    address(msg.sender),
                    block.timestamp
                );

                _amountOutToken2 = amountsOut[amountsOut.length - 1];
            } else {
                _amountOutToken2 = _secondTokenAmount;
                IERC20(_secondToken).transfer(msg.sender, _secondTokenAmount);
            }
        }

        return (_amountOutToken1, _amountOutToken2);
    }
```

Listing 3.3: Nodes_::split()

Note it shares the same issue in the `swapTokens()` routine in the `Nodes_` contract.

Recommendation Add proper access control to the above routines.

Status Per the discussion with the team, they decide to leave it as is.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: ZRX.sol

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 are safe versions of `transferFrom()/approve()` as well, i.e., `safeTransferFrom()/safeApprove()`.

In the following, we show the `sendToWallet()` routine in the `Nodes` contract. If the `ZRX` token is supported as `_token`, the unsafe version of `_token.transfer(_user, _amount)` (line 279) may proceed without a revert for transfer failure. Because it returns `false` for failure in the `ZRX` token contract's `transfer()/transferFrom()` implementation.

```
271     function sendToWallet(
272         address _user,
273         IERC20 _token,
274         uint256 _amount
275     ) public nonReentrant onlyOwner returns (uint256 amount) {
```

```
        uint256 _userBalance = getBalance(_user, _token);
        require(_userBalance >= _amount, 'Insufficient balance.');

        _token.transfer(_user, _amount);

        decreaseBalance(_user, address(_token), _amount);

        emit SendToWallet(address(_token), _amount);
        return _amount;
    }
```

Listing 3.5: Nodes::sendToWallet()

Similar violations can be found in routines: `recoverAll()/split()/swapTokens()/liquidate()`, etc.

In the following, we continue to examine the `approve()` routine and related 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 3.6: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

In the following `Nodes::liquidate()` routine, it's suggested to reduce the allowance to 0 before setting the new allowance (line 466).

```
    function liquidate(
        address _user,
        IERC20[] memory _tokens,
        uint256[] memory _amounts,
        address _tokenOutput
    ) public nonReentrant onlyOwner returns (uint256 amountOut) {
        uint256 amount;
        for (uint256 _i = 0; _i < _tokens.length; _i++) {
            address tokenInput = address(_tokens[_i]);
            uint256 amountInput = _amounts[_i];
            uint256 userBalance = getBalance(_user, IERC20(tokenInput));
            require(userBalance >= amountInput, 'Insufficient Balance.');

            uint256 _amountOut;
            if (tokenInput != _tokenOutput) {
                IERC20(tokenInput).approve(address(router), amountInput);
                ...
                IERC20(_tokenOutput).transfer(_user, _amountOut);
                amount += _amountOut;
            } else {
                IERC20(_tokenOutput).transfer(_user, amountInput);

                amount += amountInput;
            }

            decreaseBalance(_user, tokenInput, amountInput);
        }

        emit Liquidate(_tokenOutput, amount);
        return amount;
    }
```

Listing 3.7: Nodes::liquidate()

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `approve()/transfer()/transferFrom()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been fixed by the following commits: `70da7b0` and `461a51f`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the Tortle Ninja protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., update the callFee and treasury address). In the following, we examine the privileged owner account and the related privileged accesses in current contracts.

Specially, the privileged functions in the Nodes contract allow the owner to handle user assets deposited in this contract at any time. For example, the owner can deposit any amounts of user assets to UniswapV2, or withdraw any amount of user LP tokens from UniswapV2, etc.

```
    function depositOnLp(
        address user,
        address lpToken,
        address token0,
        address token1,
        uint256 amount0,
        uint256 amount1
    ) external nonReentrant onlyOwner returns (uint256) {
        require(amount0 <= getBalance(user, IERC20(token0)), 'DepositOnLp: Insufficient
            token0 funds.');
        require(amount1 <= getBalance(user, IERC20(token1)), 'DepositOnLp: Insufficient
            token1 funds.');
        (uint256 amount0f, uint256 amount1f, uint256 lpRes) = _addLiquidity(token0,
            token1, amount0, amount1);
        userLp[lpToken][user] += lpRes;
        decreaseBalance(user, address(token0), amount0f);
        decreaseBalance(user, address(token1), amount1f);
        return lpRes;
    }

    function withdrawFromLp(
    address user,
    string[] memory _arguments,
    uint256 amount
) external nonReentrant onlyOwner returns (uint256 amountTokenDesired) {
    _wffot memory args;
    args.lpToken = StringUtils.parseAddr(_arguments[1]);
    args.token0 = StringUtils.parseAddr(_arguments[3]);
    args.token1 = StringUtils.parseAddr(_arguments[4]);
```

```
    args.tokenDesired = StringUtils.parseAddr(_arguments[5]);
    args.amountTokenDesiredMin = StringUtils.safeParseInt(_arguments[6]);

    require(amount <= userLp[args.lpToken][user], 'WithdrawFromLp: Insufficient funds.')
        ;
    userLp[args.lpToken][user] -= amount;
    amountTokenDesired = _withdrawLpAndSwap(user, args, amount);
}
```

Listing 3.8: Example Privileged Operation in `Nodes.sol`

Moreover, the privileged functions in the `TortleFarmingStrategy` contract allow the `owner` to update the `callFee` which is the fee rate used to reward the caller, and update the `treasury` address which is used to receive protocol fee.

```
    function updateCallFee(uint256 _callFee) external onlyOwner returns (bool) {
        callFee = _callFee;
        treasuryFee = PERCENT_DIVISOR - callFee;
        emit FeesUpdated(callFee, treasuryFee);
        return true;
    }

    function updateTreasury(address newTreasury)
        external
        onlyOwner
        returns (bool)
    {
        treasury = newTreasury;
        return true;
    }
```

Listing 3.9: Example Privileged Operations in `TortleFarmingStrategy.sol`

There are still other privileged routines not listed here. We point out that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Making the above privileges explicit among protocol users.

Status This issue has been confirmed by the team. And the team clarify that they will create a governance mechanism to manage the privileges.

3.5 Lack of Slippage Control in Tortle

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple contracts`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

The `swap()` routine in the `TortleFarmingStrategy` contract is a helper routine that helps user to swap a specific amount of token to the target token following the given swap path.

To elaborate, we show below the code snippet of the `swap()` routine. It completes the swap by calling the `UniswapV2Router02::swapExactTokensForTokensSupportingFeeOnTransferTokens()` routine which supports an input parameter that indicates the minimum amount of the target token to be received. If this minimum amount is not met, the swap will revert to protect user assets from potential loss.

```
251     funcwtion swap(uint256 _amount, address[] memory _path) internal {
252         if (_path.length < 2   _amount == 0) {
253             return;
254         }
255         IERC20(_path[0]).safeApprove(uniRouter, 0);
256         IERC20(_path[0]).safeApprove(uniRouter, _amount);
257         IUniswapV2Router02(uniRouter)
258             .swapExactTokensForTokensSupportingFeeOnTransferTokens(
259                 _amount,
260                 0,
261                 _path,
262                 address(this),
263                 block.timestamp
264             );
265     }
```

Listing 3.10: TortleFarmingStrategy :: swap()

However, it comes to our attention that there is no slippage control in place (line 260), 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 user assets.

Status The issue has been fixed by the following commit: `e9c66f7`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Tortle Ninja` protocol, which is a no-code `DeFi` meta aggregator that allows the creation of `Tortle Combo Recipes`, a combination of `DeFi` operations that enable you to create, shoot, repeat, and copy complex investment strategies. And `Tortle` runs natively on `Fantom Opera` which is by far the most reliable, open, and fast network. The current code base is well organized and those identified issues are promptly confirmed and addressed.

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

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

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

[4] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.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.