SMART CONTRACT AUDIT REPORT

for

Thetanuts

Prepared By: Xiaomi Huang

PeckShield

May 28, 2022

Document Properties

Client	Thetanuts
Title	Smart Contract Audit Report
Target	Thetanuts
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 28, 2022	Xuxian Jiang	Final Release
1.0-rc1	May 14, 2022	Xuxian Jiang	Release Candidate #1

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 `Thetanuts` 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 audited protocol 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 Thetanuts

`Thetanuts Finance` is a new protocol that helps users to access crypto structured products on multiple decentralised networks to generate a return for their portfolio. The provided `Thetanuts strategies` enable users to earn a high base yield on their assets. In particular, users will hold a token, long that position, desire to earn more yield in that token that they are long in. This audit also includes the governance to decentralise control, and a lending market to improve capital efficiency. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Thetanuts

Item	Description
Name	Thetanuts
Website	https://thetanuts.finance
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 28, 2022

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

- https://github.com/ezoia-com/thetanuts_v1.git (16ea8dd)

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

- https://github.com/ezoia-com/thetanuts_v1.git (f54af1c)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	3	■ ■ ■
Informational	0	
Total	4	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key Thetanuts Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Reentrancy Risk in ConcRouter	Coding Practices	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-003	Low	Improved Validation on Function Arguments	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Potential Reentrancy Risk in ConcRouter

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: ConcRouter
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `ConcRouter` as an example, the `swapETHForExactTokens()` function (see the code snippet below) is provided to externally call a token contract to transfer assets and return the extra funds back to the caller. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 148) needs to start before returning the extra funds back to the user. In this particular case, if the external caller has certain hidden logic, it may manipulate the swap rate to affect the token amount after conversion within the same entry function.

```
141     function swapETHForExactTokens(uint amountOut, address[] calldata path, address to,
            uint deadline) external payable returns (uint[] memory amounts) {
142         address[] memory concPath = getConcPath(path);
```

```
        uint amountIn = ROUTER.getAmountsIn(amountOut, concPath)[0];
        require(msg.value >= amountIn, "UniswapV2Router: EXCESSIVE_INPUT_AMOUNT");
        address payable weth = payable(WETH);
        weth.call{value: amountIn}("");
        if (msg.value > amountIn) msg.sender.call{value: msg.value - amountIn}("");
        amounts = _swapTokensForExactTokens(amountOut, amountIn, path, to, deadline);
        ERC20 ogOutAsset = ERC20(path[path.length-1]);
        ogOutAsset.safeTransfer(msg.sender, amounts[concPath.length-1]);
    }
```

Listing 3.1: `ConcRouter::swapETHForExactTokens()`

Recommendation Revise the above `swapETHForExactTokens()` logic by returning the extra funds after swapping the intended tokens.

Status This issue has been fixed in the following commit: 657b1d6.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [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 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 3.2: 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 is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `swapTokensForExactTokens()` routine in the `V3Proxy` contract. If the `USDT` token is supported as `ogInAsset`, the unsafe version of `ogInAsset.transfer(msg.sender, ogInAsset.balanceOf(address(this)))` (line 129) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
121     function swapTokensForExactTokens(uint amountOut, uint amountInMax, address[]
            calldata path, address to, uint deadline) external returns (uint[] memory
            amounts) {
122         require(path.length == 2);
123         ERC20 ogInAsset = ERC20(path[0]);
124         ogInAsset.safeTransferFrom(msg.sender, address(this), amountInMax);
125         ogInAsset.approve(address(ROUTER), amountInMax);
126         amounts = new uint[](2);
127         amounts[0] = ROUTER.exactOutputSingle(ISwapRouter.ExactOutputSingleParams(path
                [0], path[1], 3000, msg.sender, deadline, amountOut, amountInMax, 0));
128         amounts[1] = amountOut;
129         ogInAsset.transfer(msg.sender, ogInAsset.balanceOf(address(this)));
130         emit Swap(msg.sender, path[0], path[1], amounts[0], amounts[1]);
131     }
```

Listing 3.3: `V3Proxy::swapTokensForExactTokens()`

In the meantime, we notice this issue is also applicable to other routines, including `_swapTokensForExactTokens()/_swapExactTokensForTokens()` in `ConcRouter`. For the `safeApprove()` support, we need to apply twice: the first time resets the allowance to 0 and the second time sets the intended allowance amount.

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

Status This issue has been fixed in the following commits: 657b1d6, a0a384e, and ff29a2b

3.3 Improved Validation on Function Arguments

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `V3Proxy`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `Thetanuts Finance` protocol has the `V3Proxy` contract that is developed to facilitate the token swaps, including the common functions `swapTokensForExactTokens()`, `swapETHForExactTokens()`, `swapTokensForExactETH()`, and `swapExactTokensForETH()`. While reviewing this set of functions, we notice a specific one can be improved.

To elaborate, we show below the related `swapExactETHForTokens()` function. As the name indicates, this function is proposed to swap the given `Ether` to the target token. However, it comes to our attention that this routine can be improved to enforce the given path truly reflects the intended token swaps.

```
    function swapExactETHForTokens(uint amountIn, uint amountOutMin, address[] calldata path, address to, uint deadline) payable external returns (uint[] memory amounts) {
        require(path.length == 2);
        amounts = new uint[](2);
        amounts[0] = amountIn;
        amounts[1] = ROUTER.exactInputSingle{value: msg.value}(ISwapRouter.ExactInputSingleParams(path[0], path[1], 3000, msg.sender, deadline, amountIn, amountOutMin, 0));
        emit Swap(msg.sender, path[0], path[1], amounts[0], amounts[1]);
    }
```

Listing 3.4: `V3Proxy::swapExactETHForTokens()`

In particular, it is helpful to validate the given path with the following requirement: `require(path[0] = WETH && amountIn == msg.value)`, which in essence validates the given token is the native `Ether` and the given `amountIn` is consistent with `msg.value`.

Recommendation Revise the above `swapExactETHForTokens()` function to validate the give arguments.

Status This issue has been fixed in the following commit: 657b1d6.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Thetanuts Finance protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the protocol-wide operations (e.g., configure parameters and execute privileged operations). They also have the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that this privileged account needs to be scrutinized. In the following, we examine their related privileged accesses in current protocol.

```
    function emergencyWithdraw(IERC20 token) external {
        require(msg.sender == owner, "Not owner");
        token.safeTransfer(msg.sender, token.balanceOf( address(this) ) );  // msg.
            sender has been Required to be owner
        emit AdminAction(3);
    }

    /*
      Set allowInteractions - owner can arbitrarily stop deposits and withdrawals.
    */
    function setAllowInteraction(bool _flag) external {
      require(msg.sender == owner, "Not owner");
      allowInteractions = _flag;
    }

    /*
      Set validator - owner can assign a validator to sign winning MAKER's bids
    */
    function setValidator(address newValidator) external {
        require(msg.sender == owner, "Not owner");
        validator = newValidator;
    }
```

Listing 3.5: Example Privileged Operations in CommonV1

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract 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 Promptly transfer the administrative privileges 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 The issue has been mitigated as the owner for Vaults will be set to OwnerProxy_*.sol. This gates the excess powers given to EOA into three tiers, with the higher tiers assigned to multisig, then eventually to a DAO.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Thetanuts Finance` protocol, which is a new protocol that helps users to access crypto structured products on multiple decentralised networks to generate a return for their portfolio. This audit also includes the governance to decentralise control, and a lending market to improve capital efficiency. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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

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

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