SMART CONTRACT AUDIT REPORT

for

TheUnfettered Protocol

Prepared By: Yiqun Chen

PeckShield

February 12, 2022

Document Properties

Client	TheUnfettered
Title	Smart Contract Audit Report
Target	TheUnfettered
Version	1.0
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 12, 2022	Yiqun Chen	Final Release
1.0-rc	February 8, 2022	Yiqun Chen	Release Candidate

Contact

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

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

Contents

1 Introduction

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

1.1 About TheUnfettered

`TheUnfettered` protocol is equipped with the staking support, which allows users to earn rewards by simply depositing their tokens into the staking pools. The user needs to wait at least one month to unstake their staked tokens with their rewards. By design, the rewards for the user will be calculated and recorded when they stake or unstake their tokens. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of TheUnfettered

Item	Description
Name	TheUnfettered
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 12, 2022

In the following, we show the contract file and the MD5/SHA checksum value of the contract file:

- File: TheUnfettered.rar
- MD5: 62cc74cf3929631b783e9a8143fcee3b

- SHA: 74684e9016b683cd31f67dfe5fcab1af346645c27916dbe937171d07218434ce

And this is the contract file and its MD5/SHA checksum value after all fixes for the issues found in the audit have been checked in:

- File: TheUnfetteredV5.rar
- MD5: b0b3746e786e3f66ad62985e36ab68ae
- SHA: 0b7fb6c7048524a3542f741d423ef8e066d17d7a6c92de8e976827a4eb05c569

1.2 About PeckShield

PeckShield Inc. [10] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [9]:

- 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 checklist of 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) [8], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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

Table 1.3: The Full Audit Checklist

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

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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business 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.

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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `TheUnfettered` 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 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	3	■ ■ ■
Low	1	■
Informational	1	■
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 3 medium-severity vulnerabilities, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key TheUnfettered Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Stake Amount In stake()	Business Logic	Fixed
PVE-002	Informational	Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-003	Low	Timely Update Reward Upon stakePercentage Change	Business Logic	Fixed
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-005	Medium	Improper stakeDate Updated In unstake()	Business Logic	Fixed

Besides 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 Improper Stake Amount In stake()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Staking`
- Category: Business Logic [7]
- CWE subcategory: CWE-770 [3]

Description

`TheUnfettered` protocol allows users to stake specified tokens, and earn the same kind of token as rewards. The rewards for the user will be calculated and recorded when the `stake()/unstake()` function is invoked. However, we notice the transfer/record of the stake amount needs to be improved.

To elaborate, we show below the related code snippet of the `stake()` routine. In the `stake()` routine, rewards will be calculated based on the previous staked amount of the user (lines 160-161). After that, it transfers the tokens from the user to the contract. However, the amount transferred is the desired amount of the user with the additional rewards amount added. Also, the previous staked amount will be replaced with this amount.

```
154     function stake(uint256 _amount) public whenNotPaused{
155         require(_amount > 0,"Amount must be greater than Zero.");
156         require(tokenContract.balanceOf(msg.sender) >= _amount,"Amount cannot be greater
                 than your balance.");
157         uint256 _newAmount = _amount;
158         if(stakes[msg.sender].amount > 0){
159             uint256 _dateDiff = block.timestamp.sub(stakes[msg.sender].stakeDate);
160             _newAmount = stakes[msg.sender].amount.mul(stakePercentage).mul(_dateDiff).
                    div(3153600000);
161             _newAmount = _newAmount.add(_amount);
162         }
163         uint256 _scaledAmount = _newAmount.mul(uint256(10) ** tokenContract.decimals());
164         require(tokenContract.transferFrom(msg.sender,address(this),_scaledAmount),"
                Token Transfer to Contract failed.");
165         stakes[msg.sender].amount = _newAmount;
```

```
        stakes[msg.sender].stakeDate = block.timestamp;
    }
```

Listing 3.1: `Staking::stake()`

Recommendation Transfer right amount of tokens from the user. And properly update the staked amount of the user.

Status This issue has been fixed as suggested.

3.2 Meaningful Events For Important State Changes

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [2]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `Staking` contract as an example. While examining the events that reflect the privileged accounts (i.e.g., `ceoAddress`, `ctoAddress`, `cfoAddress`, `advisorAddress`) changes, we notice there is a lack of emitting related events to reflect important state changes. In the following, we list below related functions.

```
    function changeCeoAddress(address _newAddress) public onlyCeo{
        ceoAddress = _newAddress;
    }

    function changeCtoAddress(address _newAddress) public onlyCto{
        ctoAddress = _newAddress;
    }

    function changeCfoAddress(address _newAddress) public onlyCfo{
        cfoAddress = _newAddress;
    }

    function changeAdvisorAddress(address _newAddress) public onlyAdvisor{
        advisorAddress = _newAddress;
```

```
    }
```

Listing 3.2: `Staking`

Additionally, there also exists same functions in the `Manager/Token` contracts, which are lack of meaningful events emitted.

Recommendation Properly emit the related events when the above-mentioned functions are being invoked.

Status This issue has been fixed as suggested.

3.3 Timely Update Reward Upon stakePercentage Change

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Staking`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, `TheUnfettered` protocol implements an incentive mechanism that rewards the staking of the supported asset. The rewards for the user will be calculated and recorded when the `stake()/unstake()` functions are invoked. However, the reward rate (i.g., `stakePercentage`) can be dynamically configured via a specific routine `changeStakePercentage()`.

When analyzing the specific routine, we notice the need of timely calculating and recording the reward distribution of all users. However, in the current implementation, one possible approach to solve this problem is to remove the `changeStakePercentage()` function once the staking event starts.

```
  function changeStakePercentage(uint16 _newPercentage) public isApprovalUnlocked{
        stakePercentage = _newPercentage;
        lockApproval();
    }
```

Listing 3.3: Staking::changeStakePercentage()

Recommendation Disable the `changeStakePercentage()` function once the staking event starts.

Status The issue has been fixed as suggested.

3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In TheUnfettered protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., pause/unpause the protocol and withdraw staked tokens). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function withdrawBalance (uint256 _amount) public isApprovalUnlocked onlyManager{
        uint256 scaledAmount = _amount.mul(uint256(10) ** tokenContract.decimals());
        require(tokenContract.transfer(ceoAddress, scaledAmount));
        lockApproval();
    }

    function withdrawTotalBalance () public isApprovalUnlocked onlyManager{
        uint256 scaledAmount = getTotalBalance();
        require(tokenContract.transfer(ceoAddress, scaledAmount));
        lockApproval();
    }

    function pause() public isApprovalUnlocked onlyManager{
        _pause();
        lockApproval();
    }
    function unpause() public isApprovalUnlocked onlyManager{
        _unpause();
        lockApproval();
    }
```

Listing 3.4: Staking::withdrawBalance()/withdrawTotalBalance()/pause()/unpause()

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of TheUnfettered design.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed.

3.5 Improper stakeDate Updated In unstake()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Staking`
- Category: Business Logic [7]
- CWE subcategory: CWE-770 [3]

Description

In the design of `TheUnfettered` protocol, the user needs to wait at least one month to unstake their staked tokens with rewards. However, it comes to our attention that the `stakeDate` of the user is improperly updated.

To elaborate, we show below the related `unstake()` routine. The `unstake()` routine is used for unstaking the staked tokens for the user. The reward is also calculated based on the current staked amount of the user in this routine. The remaining amount of staked tokens will be updated in the record. However, the `stakeDate` is updated to the `block.timestamp`. It means that if the user attempts to unstake part of staked tokens, he/she must wait another month to unstake another part.

```
    function unstake(uint256 _amount) public whenNotPaused{
        require(_amount > 0,"Amount must be greater than Zero.");
        require(stakes[msg.sender].amount > 0,"Stake Amount must be greater than Zero.");
        require(block.timestamp >= stakes[msg.sender].stakeDate.add(2592000),"You must wait at least a month to unstake.");
        uint256 _dateDiff = block.timestamp.sub(stakes[msg.sender].stakeDate);
        uint256 _totalAmount = stakes[msg.sender].amount.add(stakes[msg.sender].amount.mul(stakePercentage).mul(_dateDiff).div(3153600000));
        require(_totalAmount >= _amount,"Amount cannot be greater than your stake.");
        uint256 _scaledAmount = _amount.mul(uint256(10) ** tokenContract.decimals());
        require(tokenContract.transfer(msg.sender,_scaledAmount),"Token Transfer to Contract failed.");
        stakes[msg.sender].amount = _totalAmount.sub(_amount);
        stakes[msg.sender].stakeDate = block.timestamp;
    }
```

Listing 3.5: `Staking::unstake()`

Recommendation The `stakeDate` should not be updated in the `unstake()` function.

Status The issue has been fixed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `TheUnfettered` protocol, which provides an incentive mechanism that rewards the staking of the supported asset with reward tokens. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[3] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

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

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

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

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