



SMART CONTRACT AUDIT REPORT

for

CRYPTOHEROES



Prepared By: Shuxiao Wang

PeckShield
February 05, 2021

Document Properties

Client	CryptoHeroes
Title	Smart Contract Audit Report
Target	CryptoHeroes
Version	1.0
Author	Huaguo Shi
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	February 05, 2021	Huaguo Shi	Final Release
1.0-rc	January 29, 2021	Huaguo Shi	Release Candidate

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	4
1.1	About CryptoHeroes	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	9
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	ERC20 Compliance Checks	12
4	ERC721 Compliance Checks	15
5	Detailed Results	17
5.1	Suggested Adherence of Checks-Effects-Interactions	17
5.2	Recommended Explicit Pool Validity Checks	19
5.3	Timely massUpdatePools During Pool Weight Changes	21
5.4	Trust Issue of Admin Keys	22
6	Conclusion	24
	References	25

1 | Introduction

Given the opportunity to review the design document and related source code of the **CryptoHeroes** smart contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract exhibits no ERC20 and ERC721 compliance issues or security concerns. This document outlines our audit results.

1.1 About CryptoHeroes

CryptoHeroes is an innovative DeFi project that combines Yield Farming and NFT Market. NFT serves as a unique opportunity to start earning passive income for staking the most popular DeFi tokens on the market. The amount of income earned for staking depends on the purchased pass (NFT) to the world of decentralized farming. The amount of passive income depends on the price of the NFT card, which opens up to the investor the choice of an investment product and its profitability. This model was first implemented on the DeFi market and guarantees the client passive profitability in the range of the NFT indicated in his pass. The NFT Market is mainly for investors who prefer collecting rare goods that do nothing other than collecting.

The basic information of CryptoHeroes is as follows:

Table 1.1: Basic Information of CryptoHeroes

Item	Description
Issuer	CryptoHeroes
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	February 05, 2021

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

this audit:

- <https://github.com/heroescrypto/cryptoheroes> (6cb0eaf)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the 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.

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

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
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section 3)
ERC721 Compliance Checks	Compliance Checks (Section 4)
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

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.
- ERC20 Compliance Checks: We next manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- ERC721 Compliance Checks: We also validate whether the implementation logic of the audited smart contract(s) follows the standard ERC721 specification and other best practices.
- 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.

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.

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.



2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the CryptoHeroes. 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 ERC20 and ERC721 standards-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	
Low	1	
Informational	1	
Total	4	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 and ERC721 specification and other known best practices, and validate their compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20/ERC721 compliance checks are reported in Section 3 and Section 4, respectively. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 5.

2.2 Key Findings

Overall, no ERC20 and ERC721 compliance issue was found and our detailed checklist can be found in Section 3 and Section 4. Also, there is no critical or high severity issue, although the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key CryptoHeroes Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Confirmed
PVE-002	Informational	Recommended Explicit Pool Validity Checks	Security Features	Confirmed
PVE-003	Medium	Timely massUpdatePools During Pool Weight Changes	Business Logic	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 and Section 4 for our detailed compliance checks and Section 5 for elaboration of reported issues.

3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `view-only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited CryptoHeroes. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted

ERC20 specification.

Table 3.2: Key State-Changing Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to <i>address(0x0)</i> when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on <code>transfer()/transferFrom()</code> calls	—
Rebasing	The <code>balanceOf()</code> function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	—

4 | ERC721 Compliance Checks

The ERC721 standard for non-fungible tokens, also known as deeds. Inspired by the ERC-20 token standard, the ERC721 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC721-compliant. Naturally, we examine the list of necessary API functions defined by the ERC721 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 4.1: Basic `view-only` Functions Defined in The ERC721 Specification

Item	Description	Status
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
ownerOf()	Is declared as a public view function	✓
	Returns the address of the owner of the NFT	✓
getApproved()	Is declared as a public view function	✓
	Reverts while ' <code>_tokenId</code> ' does not exist	✓
	Returns the approved address for this NFT	✓
isApprovedForAll()	Is declared as a public view function	✓
	Returns a boolean value which check ' <code>_operator</code> ' is an approved operator	✓

Our analysis shows that there is no ERC721 inconsistency or incompatibility issue found in the audited CryptoHeroes. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 4.1) and key `state-changing` functions (Table 4.2) according to the widely-adopted ERC721 specification.

Table 4.2: Key State-Changing Functions Defined in The ERC721 Specification

Item	Description	Status
safeTransferFrom()	Is declared as a public function	✓
	Reverts while 'to' refers to a smart contract and not implement IERC721Receiver-onERC721Received	✓
	Reverts unless 'msg.sender' is the current owner, an authorized operator, or the approved address for this NFT	✓
	Reverts while 'tokenId' is not a valid NFT	✓
	Reverts while 'from' is not the current owner	✓
	Reverts while transferring to zero address	
	Emits Transfer() event when tokens are transferred successfully	✓
transferFrom()	Is declared as a public function	✓
	Reverts unless 'msg.sender' is the current owner, an authorized operator, or the approved address for this NFT	✓
	Reverts while 'tokenId' is not a valid NFT	✓
	Reverts while 'from' is not the current owner	✓
	Reverts while transferring to zero address	
	Emits Transfer() event when tokens are transferred successfully	✓
approve()	Is declared as a public function	✓
	Reverts unless 'msg.sender' is the current owner, an authorized operator, or the approved address for this NFT	✓
	Emits Approval() event when tokens are approved successfully	✓
setApprovalForAll()	Is declared as a public function	✓
	Reverts while not approving to caller	✓
	Emits ApprovalForAll() event when tokens are approved successfully	✓
Transfer() event	Is emitted when tokens are transferred	✓
Approval() event	Is emitted on any successful call to approve()	✓
ApprovalForAll() event	Is emitted on any successful call to setApprovalForAll()	✓

5 | Detailed Results

5.1 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: CryptoHeroesUniverse
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

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 are several occasions the `checks-effects-interactions` principle is violated. Using the `CryptoHeroesUniverse` as an example, the `deposit()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 195) starts before effecting the update on internal states (lines 196–199), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `deposit()` function.

```
173 // Deposit LP tokens to Contract for cherooes allocation.
174 function deposit(uint256 _pid, uint256 _amount) public {
175
176
177     PoolInfo storage pool = poolInfo[_pid];
```

```

178     UserInfo storage user = userInfo[_pid][msg.sender];
179
180     updatePool(_pid);
181
182     if (pool.NFTisNeeded == true)
183     {
184         require(pool.acceptedNFT.balanceOf(address(msg.sender))>0,"requires NTF token!")
185         ;
186     }
187
188     if (user.amount > 0) {
189         uint256 pending = user.amount.mul(pool.accCheroesPerShare).div(1e12).sub(user.
190             rewardDebt);
191         if (pending > 0) {
192             safeCheroesTransfer(msg.sender, pending);
193         }
194     }
195
196     if (_amount > 0) {
197         pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
198         user.amount = user.amount.add(_amount);
199     }
200
201     user.rewardDebt = user.amount.mul(pool.accCheroesPerShare).div(1e12);
202
203     emit Deposit(msg.sender, _pid, _amount);
204 }

```

Listing 5.1: CryptoHeroesUniverse::deposit()

Another similar violation can be found in the `withdraw()` routine within the same contract.

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions best practice.

Status This issue has been confirmed.

5.2 Recommended Explicit Pool Validity Checks

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: CryptoHeroesUniverse
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

The reward mechanism in CryptoHeroes relies on the pool contract for a number of tasks, including the pool management, staking/unstaking support, as well as the reward distribution to various pools and stakers. In the following, we use the `CryptoHeroesUniverse` contract as an example and show the key pool data structure. Note all added pools are maintained in an array `poolInfo`.

```

45 // Info of each pool.
46 struct PoolInfo
47 {
48     IERC20 lpToken;           // Address of LP token contract.
49     bool NFTisNeeded;         // need NFT or not
50     IERC721 acceptedNFT;      // What NFTs accepted for staking.
51     uint256 allocPoint;       // How many allocation points assigned to this pool. POBs
                                // to distribute per block.
52     uint256 lastRewardBlock;  // Last block number that POBs distribution occurs.
53     uint256 accCheroesPerShare; // Accumulated Cheroes per share, times 1e12. See below.
54 }

```

Listing 5.2: `CryptoHeroesUniverse::PoolInfo`

When there is a need to add a new pool, set a new `allocPoint` for an existing pool, stake (by depositing the supported assets), unstake (by redeeming previously deposited assets), query pending cheroes rewards, there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range `[0, poolInfo.length-1]`. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say `validatePool`. This new modifier essentially ensures the `given_pool_id` or `_pid` indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not!

```

173 // Deposit LP tokens to Contract for cheroes allocation.
174 function deposit(uint256 _pid, uint256 _amount) public {
175
176
177     PoolInfo storage pool = poolInfo[_pid];
178     UserInfo storage user = userInfo[_pid][msg.sender];
179

```

```

180     updatePool(_pid);
181
182     if(pool.NFTisNeeded == true)
183     {
184         require(pool.acceptedNFT.balanceOf(address(msg.sender))>0,"requires NTF token!")
185         ;
186     }
187
188     if (user.amount > 0) {
189         uint256 pending = user.amount.mul(pool.accCheroesPerShare).div(1e12).sub(user.
190             rewardDebt);
191         if(pending > 0) {
192             safeCheroesTransfer(msg.sender, pending);
193         }
194     }
195
196     if(_amount > 0) {
197         pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
198         user.amount = user.amount.add(_amount);
199     }
200
201     user.rewardDebt = user.amount.mul(pool.accCheroesPerShare).div(1e12);
202
203     emit Deposit(msg.sender, _pid, _amount);
204 }

```

Listing 5.3: CryptoHeroesUniverse::deposit()

We highlight that there are a number of functions that can be benefited from the new pool-validating modifier, including set(), deposit(), withdraw(), and updatePool().

Recommendation Apply necessary sanity checks to ensure the given _pid is legitimate. Accordingly, a new modifier validatePool can be developed and appended to each function in the above list.

```

173 modifier validatePool(uint256 _pid) {
174     require(_pid < poolInfo.length, "chef: pool exists?");
175     _;
176 }
177
178 // Deposit LP tokens to Contract for cheroes allocation.
179 function deposit(uint256 _pid, uint256 _amount) public validatePool(_pid) {
180
181
182     PoolInfo storage pool = poolInfo[_pid];
183     UserInfo storage user = userInfo[_pid][msg.sender];
184
185     updatePool(_pid);
186
187     if(pool.NFTisNeeded == true)
188     {

```

```

189     require(pool.acceptedNFT.balanceOf(address(msg.sender))>0,"requires NTF token!")
190     ;
191 }
192 if (user.amount > 0) {
193     uint256 pending = user.amount.mul(pool.accCheroesPerShare).div(1e12).sub(user.
        rewardDebt);
194     if(pending > 0) {
195         safeCheroesTransfer(msg.sender, pending);
196     }
197 }
198
199 if(_amount > 0) {
200     pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
201     user.amount = user.amount.add(_amount);
202 }
203
204 user.rewardDebt = user.amount.mul(pool.accCheroesPerShare).div(1e12);
205
206 emit Deposit(msg.sender, _pid, _amount);
207 }

```

Listing 5.4: CryptoHeroesUniverse::deposit()

Status This issue has been confirmed.

5.3 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: CryptoHeroesUniverse
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 5.2, the CryptoHeroes protocol provides incentive mechanisms that reward the staking of supported assets with `lpTokens`. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```

111 // Update the given pool's CHEROES allocation point. Can only be called by the owner.
112 function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
113     if (_withUpdate) {
114         massUpdatePools();
115     }
116     totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
117     poolInfo[_pid].allocPoint = _allocPoint;
118 }

```

Listing 5.5: CryptoHeroesUniverse::set()

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```

111 // Update the given pool's CHEROES allocation point. Can only be called by the owner.
112 function set(uint256 _pid, uint256 _allocPoint) public onlyOwner {
113     massUpdatePools();
114     totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
115     poolInfo[_pid].allocPoint = _allocPoint;
116 }

```

Listing 5.6: CryptoHeroesUniverse::set()

Status This issue has been confirmed.

5.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: CryptoHeroesUniverse
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In CryptoHeroes, the privileged account plays a critical role in governing and regulating the system-wide operations (e.g., pool migration). It also has the privilege to control or govern the flow of assets for staking and rewards. Our analysis shows that the privileged account needs to be scrutinized. In

the following, we examine the privileged account and their related privileged accesses in current contracts.

```

240 // Set the migrator contract. Can only be called by the owner.
241     function setMigrator(IMigratorChef _migrator) public onlyOwner {
242         migrator = _migrator;
243     }
244
245     // Migrate lp token to another lp contract. Can be called by anyone. We trust that
        migrator contract is good.
246     function migrate(uint256 _pid) public {
247         require(address(migrator) != address(0), "migrate: no migrator");
248         PoolInfo storage pool = poolInfo[_pid];
249         IERC20 lpToken = pool.lpToken;
250         uint256 bal = lpToken.balanceOf(address(this));
251         lpToken.safeApprove(address(migrator), bal);
252         IERC20 newLpToken = migrator.migrate(lpToken);
253         require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
254         pool.lpToken = newLpToken;
255     }

```

Listing 5.7: CryptoHeroesUniverse::setMigrator()

Specifically, we examine the privileged function `setMigrator()`. Notice that the privileged account is able to transfer the full `lpTokens` to a new chosen `migrator`.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract.

Status This issue has been confirmed.

6 | Conclusion

In this security audit, we have examined the CryptoHeroes design and implementation. During our audit, we first checked all respects related to the compatibility of the ERC20 and ERC721 specification and other known ERC Standard pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, four issues of varying severity were discovered and promptly confirmed by the team. In the meantime, as disclaimed in [Section 1.4](#), we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.



References

- [1] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.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: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [6] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.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>.

