SMART CONTRACT AUDIT REPORT

for

2Crazy

Prepared By: Yiqun Chen

PeckShield

December 1, 2021

Document Properties

Client	2Crazy
Title	Smart Contract Audit Report
Target	2Crazy
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 1, 2021	Shulin Bie	Final Release
1.0-rc	November 30, 2021	Shulin Bie	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 source code of the given contracts of `2Crazy`, 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 2Crazy

`2Crazy` is a revolutionary `NFT` trade platform, which greatly contributes to drive the `NFT` space to fuel mainstream adoption. In particular, `2Crazy` provides a robust and flexible `NFT` trade framework that allows third parties to integrate their own applications into it. `2Crazy` enriches the `NFT` market and also presents a unique contribution to current DeFi ecosystem.

The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of 2Crazy

Item	Description
Target	2Crazy
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 1, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. (Note that the `TokenMiddleware.sol` contract is out of our audit scope.)

- https://github.com/NFTrade/2crazy-contracts.git (b9915dc)

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

- https://github.com/NFTrade/2crazy-contracts.git (f017e20)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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

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

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) [9], 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 `2Crazy` 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	1	■
Medium	1	■
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 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 high-severity vulnerability, 1 medium-severity vulnerability, and 3 low-severity vulnerabilities.

Table 2.1: Key 2Crazy Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Reentrancy Risk In TwoCrazy	Time and State	Fixed
PVE-002	High	Improper Logic Of withdrawFunds()	Business Logic	Fixed
PVE-003	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Low	Incompatibility With Deflationary/Re-basing Tokens	Business Logic	Confirmed
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Potential Reentrancy Risk In TwoCrazy

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `TwoCrazy`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

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` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

In the `TwoCrazy` contract, we notice the `buyNFT()` routine has the potential reentrancy risk. To elaborate, we show below the code snippet of the `buyNFT()` routine in the `TwoCrazy` contract. In the `buyNFT()` routine, we notice `ERC20(feeToken).safeTransferFrom(msg.sender, address(this), currentNFT.price * amount)` (lines 873-877) is called to transfer the `feeToken` to the `TwoCrazy` contract before `purchases[msg.sender][farmIndex][NFTIndex]` and `farmNFTs[farmIndex][NFTIndex].totalPurchased` update. If the `feeToken` faithfully implements the ERC777-like standard, then the `buyNFT()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend()` and tokensReceived() hooks. Consequently, any transfer() or transferFrom() of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in ERC20(feeToken).safeTransferFrom(msg.sender, address(this), currentNFT.price * amount) (lines 873-877) before purchases[msg.sender][farmIndex][NFTIndex] and farmNFTs[farmIndex][NFTIndex].totalPurchased update. By doing so, the limit of maxUserPurchases and currentNFT.totalSupply can be bypassed, which directly undermines the assumption of the TwoCrazy design.

```
840     function buyNFT(uint256 farmIndex, uint256 NFTIndex, uint256 amount)
841         payable
842         public
843         farmExists(farmIndex)
844         farmValid(farmIndex)
845     {
846         require(!farms[farmIndex].lottery, "lottery has no buy functionality");
847         require(farmNFTs[farmIndex][NFTIndex].valid, "NFT is not valid");
848         require(block.timestamp >= farms[farmIndex].endTime, "Buy has not been started
                yet");
849         NFT memory currentNFT = farmNFTs[farmIndex][NFTIndex];
850         uint256 maxUserPurchases = getMaxPerUser(farmIndex, NFTIndex, msg.sender);
851 
852         uint256 totalUserPurchased = purchases[msg.sender][farmIndex][
853             NFTIndex
854         ];
855         address feeToken = currentNFT.feeToken;
856 
857         require(
858             (totalUserPurchased + amount) <= maxUserPurchases,
859             "Max purchases per user reached"
860         );
861         require(
862             (currentNFT.totalPurchased + amount) <= currentNFT.totalSupply,
863             "Total supply reached its limit"
864         );
865         if (
866             feeToken != address(0)
867         ) {
868             require(
869                 ERC20(feeToken).balanceOf(msg.sender) >= (currentNFT.price * amount),
870                 "Not enough balance on Fee Token to buy"
871             );
872             // transfer the feeToken to this contract
873             ERC20(feeToken).safeTransferFrom(
874                 msg.sender,
875                 address(this),
876                 currentNFT.price * amount
877             );
878 
879             feesCollected[feeToken] = feesCollected[feeToken] + (currentNFT.price *
```

```
                amount);
        } else {
            require(msg.value == (currentNFT.price * amount), 'Not enough value');
        }

        Middleware(currentNFT.contractAddress).buy(
            msg.sender,
            amount,
            farmIndex,
            NFTIndex
        );

        emit NFTBought(
            msg.sender,
            farmIndex,
            NFTIndex,
            amount,
            feeToken
        );

        purchases[msg.sender][farmIndex][NFTIndex] =
            purchases[msg.sender][farmIndex][NFTIndex] +
            amount;
        farmNFTs[farmIndex][NFTIndex].totalPurchased =
            currentNFT.totalPurchased +
            amount;
    }
```

Listing 3.1: `TwoCrazy::buyNFT()`

Note other routines, i.e., `stake()` and `withdraw()`, can also benefit from the reentrancy protection. Additionally, we believe it's a better option for the `buyNFT()` and `withdrawFunds()` routines to follow the known best practice of the `checks-effects-interactions` pattern.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been addressed in this commit: `fb43fbb`.

3.2 Improper Logic Of withdrawFunds()

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `TwoCrazy`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

According to the `2Crazy` design, the `TwoCrazy` contract will likely accumulate a huge amount of assets with the increased transactions through `buyNFT()`, while the privileged `EDITOR_ROLE` account has capability to withdraw the assets with the calling of `withdrawFunds()`. While examining the logic of the `withdrawFunds()` routine, we observe there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `withdrawFunds()` routine. In the `withdrawFunds()` routine, we notice `ERC20(feeToken).transferFrom(address(this), msg.sender, feesCollected[feeToken])` is called to transfer the `feeToken` locked in `address(this)` to `msg.sender`. This is reasonable under the assumption that the `transferFrom()`'s implementation of `feeToken` supports the user spends his/her own token without approval. Otherwise, the `feeToken` locked in the `TwoCrazy` contract will be lost forever.

```
function withdrawFunds(address feeToken) public payable onlyRole(EDITOR_ROLE) {
    ERC20(feeToken).transferFrom(address(this), msg.sender, feesCollected[feeToken])
        ;
    feesCollected[feeToken] = 0;
}
```

Listing 3.2: `TwoCrazy::withdrawFunds()`

Recommendation Correct the implementation of the `withdrawFunds()` routine as below.

```
function withdrawFunds(address feeToken) public payable onlyRole(EDITOR_ROLE) {
    uint256 collectedFees = feesCollected[feeToken];
    feesCollected[feeToken] = 0;
    ERC20(feeToken).safeTransfer(msg.sender, collectedFees);
}
```

Listing 3.3: `TwoCrazy::withdrawFunds()`

Status The issue has been addressed in this commit: `fb43fbb`.

3.3 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TwoCrazy`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [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 this section, we examine the `transfer()` routine and possible 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 is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `withdraw()` routine in the `TwoCrazy` contract. If the `USDT` token is supported as `stakingToken`, the unsafe version of `require(ERC20(stakingToken).transfer(msg.sender, withdrawAmount))` (line 512) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation. We may intend to replace `require(ERC20(stakingToken).transfer(msg.sender, withdrawAmount))` (line 512) with `ERC20(stakingToken).safeTransfer(msg.sender, withdrawAmount)`.

```
471     function withdraw(uint256 farmIndex)
472     public
473     farmExists(farmIndex)
474     farmValid(farmIndex)
475     {
476         uint256 withdrawAmount = 0;
477         for (
478             uint256 i = 0;
479             i < farmStakes[farmIndex][msg.sender].length;
480             i++
481         ) {
482             uint256 lockPeriod = farmStakes[farmIndex][msg.sender][i]
483                 .lockPeriod * 86400; // days to deconds
484             uint256 secondsSinceCreated = block.timestamp -
485                 farmStakes[farmIndex][msg.sender][i].createdAt;
486 
487             if (
488                 secondsSinceCreated >= lockPeriod &&
489                 !farmStakes[farmIndex][msg.sender][i].withdrew
490             ) {
491                 withdrawAmount =
492                     withdrawAmount +
493                     farmStakes[farmIndex][msg.sender][i].amount;
494 
495                 farmStakes[farmIndex][msg.sender][i].withdrew = true;
496             }
497         }
498 
499         require(withdrawAmount > 0, "user have no tokens to withdraw");
500 
501         farms[farmIndex].totalStaked =
502             farms[farmIndex].totalStaked -
503             withdrawAmount;
504 
505         farmBalances[farmIndex][msg.sender].amount =
506             farmBalances[farmIndex][msg.sender].amount -
507             withdrawAmount;
508 
```

```
        userStakes[msg.sender] = userStakes[msg.sender] - withdrawAmount;

        // transfer back the user his stakes
        require(ERC20(stakingToken).transfer(msg.sender, withdrawAmount));

        emit Withdrew(msg.sender, farmIndex, withdrawAmount);
    }
```

Listing 3.5: `TwoCrazy::withdraw()`

Note other routines, i.e., `stake()` and `withdrawFunds()`, can be similarly improved.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()` and `transferFrom()`.

Status The issue has been addressed in this commit: `fb43fbb`.

3.4 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TwoCrazy`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

By design, the `TwoCrazy` contract is the main entry for interaction with users. In particular, one entry routine, i.e., `stake()`, accepts user deposits of the supported `stakingToken` assets. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `TwoCrazy` contract. These asset-transferring routines will work well under the assumption that the vault's internal asset balances (specified by the `farmBalances[farmIndex][msg.sender].amount`) are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
    function stake(uint256 farmIndex, uint256 amount)
    external
    farmExists(farmIndex)
    farmValid(farmIndex)
    {
        require(block.timestamp >= farms[farmIndex].startTime, "staking hasn't started
            yet");
        require(block.timestamp <= farms[farmIndex].endTime, "staking has finished");
        require(amount > 0, "staking amount must be greater than 0");
        if (farms[farmIndex].lottery) {
            require(!farmBalances[farmIndex][msg.sender].exists, "account can only stake
                once on lottery");
```

```
            require(farms[farmIndex].stakeAmount == amount, "lottery farm has a fixed stake amount");
        }
        if (farms[farmIndex].maxUsers > 0) {
            require(farms[farmIndex].maxUsers > farmAddresses[farmIndex].length, "max users reached");
        }
        // transfers the amount from the sender to the contract
        require(
            ERC20(stakingToken).transferFrom(msg.sender, address(this), amount)
        );
        // farm stakes
        farmStakes[farmIndex][msg.sender].push(
            Stake(
                amount, // amount
                block.timestamp, // createdAt
                farms[farmIndex].lockPeriod, // lockPeriod from reward
                false // withdrew
            )
        );
        // farm user balances
        farmBalances[farmIndex][msg.sender].amount =
            farmBalances[farmIndex][msg.sender].amount +
            amount;

        userStakes[msg.sender] = userStakes[msg.sender] + amount;
        // updates total staked
        farms[farmIndex].totalStaked =
            farms[farmIndex].totalStaked +
            amount;

        if (!farmBalances[farmIndex][msg.sender].exists) {
            farmBalances[farmIndex][msg.sender].exists = true;
            farmAddresses[farmIndex].push(msg.sender);
        }


        emit Staked(
            msg.sender,
            farmIndex,
            farms[farmIndex].lockPeriod,
            amount
        );
    }
```

Listing 3.6: `TwoCrazy::stake()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as stake()/withdraw(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of TwoCrazy and affects protocol-wide operation and maintenance.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in transfer() or transferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the TwoCrazy before and after the transfer() or transferFrom() is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

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

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

Status The issue has been confirmed by the team. There is no need to support deflationary/re-basing tokens.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: TwoCrazy
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the 2Crazy protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function toggleFarmValidity(uint256 farmIndex)
        external
        farmExists(farmIndex)
```

```
        onlyRole(EDITOR_ROLE)
    {
        farms[farmIndex].valid = !farms[farmIndex].valid;
        emit FarmValidityChanged(farmIndex, farms[farmIndex].valid);
    }

    ...

    function toggleNFTValidity(uint256 farmIndex, uint256 NFTIndex)
        external
        farmExists(farmIndex)
        onlyRole(EDITOR_ROLE)
    {
        farmNFTs[farmIndex][NFTIndex].valid = !farmNFTs[farmIndex][NFTIndex].valid;
        emit NFTValidityChanged(farmIndex, NFTIndex, farmNFTs[farmIndex][NFTIndex].valid
            );
    }

    ...

    function setNFTContractAddress(
        uint256 farmIndex,
        uint256 NFTIndex,
        address contractAddress
    ) external
    farmExists(farmIndex)
    onlyRole(EDITOR_ROLE) {
        farmNFTs[farmIndex][NFTIndex].contractAddress = contractAddress;
        emit NFTContractAddressChanged(farmIndex, NFTIndex, contractAddress);
    }
```

Listing 3.7: `TwoCrazy::toggleFarmValidity()&&toggleNFTValidity()&&setNFTContractAddress()`

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 the 2Crazy 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 by the team.

4 Conclusion

In this audit, we have analyzed the `2Crazy` design and implementation. As a revolutionary `NFT` trade platform, `2Crazy` is significantly contributing to drive the `NFT` space to fuel mainstream adoption. In particular, `2Crazy` provides a robust and flexible `NFT` trade framework that allows third parties to integrate their own applications into it, which presents a unique contribution to current DeFi ecosystem. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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 CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

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

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