SMART CONTRACT AUDIT REPORT

for

Dtravel NFT

Prepared By: Yiqun Chen

PeckShield

November 18, 2021

Document Properties

Client	Dtravel
Title	Smart Contract Audit Report
Target	Dtravel NFT
Version	1.0
Author	Xuxian Jiang
Auditors	Patrick Liu, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 18, 2021	Xuxian Jiang	Final Release
1.0-rc1	November 14, 2021	Xuxian Jiang	Release Candidate #1

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 4

1.1 About Dtravel 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings 9

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results 11

3.1 No Secret After Mined In getPasscode() 11

3.2 Trust Issue of Admin Keys 12

3.3 Suggested Adherence Of Checks-Effects-Interactions Pattern 13

3.4 Generation of Meaningful Events For Important State Changes 15

4 Conclusion 17

References 18

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Dtravel-NFT 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 Dtravel

Dtravel is a decentralized platform for the home-sharing economy that facilitates accommodation discovery, booking, and payments. Dtravel users can make payments with both fiat currencies and popular cryptocurrencies, including TRVL - the native utility token of the Dtravel network. Within the Dtravel ecosystem, TRVL can be used for payments, incentives and rewards, participation in Dtravel DAO governance, and to provide liquidity to decentralized exchanges for rewards. This audit covers the Dtravel membership card based on the NFT ERC721 specification.

The basic information of the Dtravel-NFT protocol is as follows:

Table 1.1: Basic Information of The Dtravel-NFT Protocol

Item	Description
Name	Dtravel
Website	https://www.dtravel.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 18, 2021

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

- https://github.com/dTravel/dtravel-nft-contract.git (83d2b39)

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

- https://github.com/dTravel/dtravel-nft-contract.git (f20a68c)

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.

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

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would 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 `Dtravel-NFT` 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	2	■ ■
Informational	1	■
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 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Dtravel NFT Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	No Secret After Mined In getPasscode()	Business Logic	Resolved
PVE-002	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-003	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Resolved
PVE-004	Informational	Meaningful Events For Important States Change	Coding Practices	Resolved

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 for details.

3 | Detailed Results

3.1 No Secret After Mined In getPasscode()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CardBase`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Dtravel-NFT` protocol allows to purchase `NFT`-based membership card with `TRVL` or native coins (`ETH` or `BNB`). Each payment requires a parameter named `passcode`, which is set by the protocol owner. The purchasing user must provide the correct `passcode` to successfully obtain a membership. It should be mentioned that the so-called `passcode`-based authentication is weak as once it is mined, everyone is able to read it. In other words, the `passcode` content becomes public.

```
function buyInNative(uint256 passcode_)
  external
  payable
  whenNotPaused
  nonReentrant
  returns (uint256)
{
  require(_passcode == passcode_, "CardManager: Wrong passcode");

  uint256 cardPriceUsdCent = _batchSaleInfo.priceUsdCent;
  require(cardPriceUsdCent > 0, "CardManager: invalid card price");

  uint256 cardPriceInNative = getCardPriceInNative();

  // Check if user-transferred amount is enough
  require(
    msg.value >= cardPriceInNative,
    "CardManager: user-transferred amount not enough"
  );
```

```
    // Mint card
    (uint256 mintedTokenId, string memory cardTokenURI) = doMintCard(
      _msgSender()
    );

    // Transfer msg.value from user wallet to beneficiary
    _beneficiary.transfer(msg.value);

    _totalNativeTokensCollected += msg.value;

    emit EventBuyInNative(_msgSender(), mintedTokenId, cardTokenURI, msg.value);

    return mintedTokenId;
  }
```

Listing 3.1: `CardBase::buyInNative()`

Note both `buyInTrvl()` and `buyInNative()` functions share the same issue.

Recommendation Revisit the above design on whether there is a need of using `passcode` for authorization check.

Status The issue has been resolved as the team clarifies that the passcode is in place to harden the access to the contract's external functions. Even in case passcode is known, it will not affect and create any harm.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Dtravel-NFT` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and sale adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. 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.

```
  function setBeneficiary(address beneficiary_) external isAuthorized {
    require(
      beneficiary_ != address(0),
```

```
      "CardManager: Invalid beneficiary_ address"
    );
    _beneficiary = payable(beneficiary_);
  }

  function setTrvlTokenPriceInUsdCent(uint256 trvlTokenPriceInUsdCent_)
    external
    isAuthorized
  {
    require(
      trvlTokenPriceInUsdCent_ > 0,
      "CardManager: Invalid TRVL token price in USD Cent"
    );

    _trvlTokenPriceInUsdCent = trvlTokenPriceInUsdCent_;
  }
```

Listing 3.2: Example Privileged Opeations in `CardManager`

If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 with the team. For the time being, the team has confirmed that these privileged functions will be called by a trusted multi-sig account, not a plain EOA account.

3.3 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CardManager`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [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].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `CardBase` as an example, the `buyInTrvl()` 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 (lines 391 – 395) starts before effecting the update on the internal state (line 397), 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 same entry function.

```
360 // Buy card in TRVL tokens
361 function buyInTrvl(uint256 passcode_)
362   external
363   whenNotPaused
364   nonReentrant
365   returns (uint256)
366 {
367   require(_passcode == passcode_, "CardManager: Wrong passcode");

369   require(
370     _trvlTokenPriceInUsdCent > 0,
371     "CardManager: TRVL token price not set"
372   );

374   uint256 cardPriceUsdCent = _batchSaleInfo.priceUsdCent;
375   require(cardPriceUsdCent > 0, "CardManager: invalid card price");

377   uint256 cardPriceInTrvlTokens = getCardPriceInTrvlTokens();

379   // Check if user balance has enough tokens
380   require(
381     cardPriceInTrvlTokens <= _trvlToken.balanceOf(_msgSender()),
382     "CardManager: user balance does not have enough TRVL tokens"
383   );

385   // Mint card
386   (uint256 mintedTokenId, string memory cardTokenURI) = doMintCard(
387     _msgSender()
388   );
```

```
    // Transfer tokens from user wallet to beneficiary
    _trvlToken.safeTransferFrom(
      _msgSender(),
      _beneficiary,
      cardPriceInTrvlTokens
    );

    _totalTrvlTokensCollected += cardPriceInTrvlTokens;

    emit EventBuyInTrvl(
      _msgSender(),
      mintedTokenId,
      cardTokenURI,
      cardPriceInTrvlTokens
    );

    return mintedTokenId;
  }
```

Listing 3.3: `CardBase::buyInTrvl()`

In the meantime, we should mention that the current implementation has taken proper prevention measures in using the `nonReentrant` modifier. However, from the code practice perspective, we still suggest to follow the `checks-effects-interactions` pattern.

Recommendation Apply necessary reentrancy prevention by following the known `checks-effects-interactions` pattern.

Status The issue has been resolved as the team considers the use of `nonReentrant` should be sufficient.

3.4 Generation of Meaningful Events For Important State Changes

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CardBase`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

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 `CardBase` contract as an example. This contract has public functions that are used to configure the `_authorizedAddressList`. While examining the events that reflect the `_authorizedAddressList` changes, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the `_authorizedAddressList` is being updated in `CardBase::grantAuthorized()`, there is no respective event being emitted to reflect the update of `authorizedAddressList` (lines 29 and 35).

```
26  function grantAuthorized(address auth_) external isOwner {
27    require(auth_ != address(0), "CardBase: invalid auth_ address ");

29    _authorizedAddressList[auth_] = true;
30  }

32  function revokeAuthorized(address auth_) external isOwner {
33    require(auth_ != address(0), "CardBase: invalid auth_ address ");

35    _authorizedAddressList[auth_] = false;
36  }
```

Listing 3.4: `CardBase::grantAuthorized()/revokeAuthorized()`

Recommendation Properly emit respective events when a new authorized entity becomes effective .

Status This issue has been fixed in the following commit: `f20a68c`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Dtravel-NFT` protocol, which implements the `Dtravel` membership card based on the `NFT ERC721` specification. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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

References

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

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

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.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: Concurrency. https://cwe.mitre.org/data/definitions/557.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.