SMART CONTRACT AUDIT REPORT

for

PNS

Prepared By: Patrick Lou

PeckShield

April 29, 2022

Document Properties

Client	PNS
Title	Smart Contract Audit Report
Target	PNS
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 29, 2022	Xiaotao Wu	Final Release
1.0-rc	April 18, 2022	Xiaotao Wu	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the PNS design document and related smart contract source code, 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 PNS

The PNS is an open, decentralized domain name system on the Polkadot blockchain. With PNS, every user can have their on-chain unique name, and resolves to their wallet account, smart contract address, NFT token, URL or IPFS address. PNS is the universal passport of Web3 ecosystem.

The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of PNS

Item	Description
Name	PNS
Website	https://www.pns.link/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 29, 2022

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

- https://github.com/pnsproject/pns-contracts.git (29d9116)

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 the 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 checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

Table 1.3: The Full Audit Checklist

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

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

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered 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 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.

2 | Findings

2.1 Summary

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

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

Table 2.1: Key PNS Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Lack Of tokenId Available Check In PNSController::setMetadataBatch()	Business Logic	Confirmed
PVE-002	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Resolved
PVE-003	Informational	Meaningful Events For Important State Changes	Coding Practices	Resolved
PVE-004	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 Lack Of tokenId Available Check In PNSController::setMetadataBatch()

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

Description

The `PNSController` contract provides a public `setMetadataBatch()` function for the privileged account (i.e., `manager`) to set metadata in batches. While examining the `setMetadataBatch()` routine, we notice the current logic is not implemented properly.

To elaborate, we show below its code snippet. It comes to our attention that there is a lack of `tokenId` available check before updating the mapping state variable `records[tokenId]` (lines 142-145). If the `tokenId` has been registered by a user, then this user's `records` corresponding to this `tokenId` will be modified.

```
136     function setMetadataBatch(uint256[] calldata data) public live onlyManager {
137         uint256 len = data.length;
138         require(len % 5 == 0, "length invalid");

140         for (uint256 i = 0; i < len; i+=5) {
141             uint256 tokenId = data[i];
142             records[tokenId].origin = data[i+1];
143             records[tokenId].expire = uint64(data[i+2]);
144             records[tokenId].capacity = uint64(data[i+3]);
145             records[tokenId].children = uint64(data[i+4]);
146         }
147         emit MetadataUpdated(data);
148     }
```

Listing 3.1: `PNSController::setMetadataBatch()`

Recommendation Add tokenId available check before updating the mapping state variable records[tokenId].

Status This issue has been confirmed.

3.2 Improved Precision By Multiplication And Division Reordering

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PNSController
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [1]

Description

SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (mul) and division (div) are involved.

In particular, we use the PNSController::totalRegisterPrice() as an example. This routine is used to calculate the total register price with the given input name and duration. And the actually rentPrice is calculated with a combination of mul/div operations (line 366). All these operations are intended for uint256. We point out that if there is a sequence of multiplication and division operations, it is always better to calculate the multiplication before the division (on the condition without introducing any extra overflows). By doing so, we can achieve better precision.

```
364     function totalRegisterPrice(string memory name, uint256 duration) view public
            override returns(uint256) {
365         uint256 price = uint256(getTokenPrice());
366         return (basePrice(name) + rentPrice(name, duration).div(86400*365)).mul(10 **
                26).div(price);
367     }
```

Listing 3.2: PNSController::totalRegisterPrice()

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in this commit: 5b9c841.

3.3 Meaningful Events For Important State Changes

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: PNS
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

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 PNS contract as an example. While examining the events that reflect the PNS dynamics, we notice there is a lack of emitting related events to reflect important state changes. Specifically, when the setContractConfig()/setName()/setNftName() are being called, there are no corresponding events being emitted to reflect the occurrence of setContractConfig()/setName()/setNftName().

```
    function setContractConfig(uint256 _writable) public onlyRoot {
        FLAGS = _writable;
    }
```

Listing 3.3: PNS::setContractConfig()

```
    function setName(
        uint256 tokenId
    ) external override writable authorised(tokenId) {
        _names[_msgSender()] = tokenId;
    }
```

Listing 3.4: PNS::setName()

```
    function setNftName(
        address nft,
        uint256 nftTokenId,
        uint256 nameTokenId
    ) external override writable authorised(nameTokenId) {
        require(IERC721Upgradeable(nft).ownerOf(nftTokenId) == _msgSender(), 'not token
            owner');
        _nft_names[nft][nftTokenId] = nameTokenId;
    }
```

Listing 3.5: PNS::setName()

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

Status This issue has been fixed in this commit: `5b9c841`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `PNSController`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `PNS` protocol, there are two privileged account, i.e., `root and manager`. These accounts play a critical role in governing and regulating the system-wide operations (e.g., change the capacity of an existing `tokenId`'s `records`, set the metadata in batches, set/update the nested mapping state variable `_records`, mint/burn `NFT` tokens, and set the key parameters for the `PNS` protocol, etc.). Our analysis shows that these privileged accounts need to be scrutinized.

In the following, we use the `SystemSettings` contract as an example and show the representative functions potentially affected by the privileges of the `root/manager` accounts.

```
    function setContractConfig(uint256 _flags, uint256 _min_length, uint256
        _min_duration, uint256 _grace_period, uint256 _default_capacity, uint256
        _capacity_price, address _price_feed) public live onlyRoot {
        FLAGS = _flags;
        MIN_REGISTRATION_LENGTH = _min_length;
        MIN_REGISTRATION_DURATION = _min_duration;
        GRACE_PERIOD = _grace_period;
        DEFAULT_DOMAIN_CAPACITY = _default_capacity;
        capacityPrice = _capacity_price;
        priceFeed = AggregatorV3Interface(_price_feed);
        emit ConfigUpdated(_flags);
    }

    function setCapacityByManager(uint256 tokenId, uint256 _capacity) public override
        live onlyManager {
        records[tokenId].capacity = uint64(_capacity);
        emit CapacityUpdated(tokenId, _capacity);
    }

    function setMetadataBatch(uint256[] calldata data) public live onlyManager {
        uint256 len = data.length;
        require(len % 5 == 0, "length invalid");
```

```
        for (uint256 i = 0; i < len; i+=5) {
            uint256 tokenId = data[i];
            records[tokenId].origin = data[i+1];
            records[tokenId].expire = uint64(data[i+2]);
            records[tokenId].capacity = uint64(data[i+3]);
            records[tokenId].children = uint64(data[i+4]);
        }
        emit MetadataUpdated(data);
    }
```

Listing 3.6: `PNSController::setContractConfig()/setCapacityByManager()/setMetadataBatch()`

```
    function mintSubdomain(address to, uint256 tokenId, string calldata name) public virtual override live authorised(tokenId) {
        uint256 originId = records[tokenId].origin;
        require(records[originId].children < records[originId].capacity, "reach subdomain capacity");
        uint256 subtokenId = _pns.mintSubdomain(to, tokenId, name);

        records[originId].children += 1;
        records[subtokenId].origin = originId;

        emit NewSubdomain(to, tokenId, subtokenId, name);
    }

    function burn(uint256 tokenId) public virtual live override {
        require(nameExpired(tokenId)  _root == _msgSender()  _pns.isApprovedOrOwner(_msgSender(), tokenId)  _pns.isApprovedOrOwner(_msgSender(), records[tokenId].origin), "not owner nor approved");
        // require subtokens cleared
        require(records[tokenId].origin != 0, "missing metadata");
        require(records[tokenId].children == 0, "subdomains not cleared");
        _pns.burn(tokenId);

        uint256 originId = records[tokenId].origin;
        if (records[originId].children > 0) {
          records[originId].children -= 1;
        }
        records[tokenId].expire = 0;
        records[tokenId].capacity = 0;
        records[tokenId].origin = 0;
    }

    function burnBatch(uint256[] calldata data) public virtual onlyManager {
        uint256 len = data.length;

        for (uint256 i = 0; i < len; i++) {
            uint256 tokenId = data[i];
            uint256 originId = records[tokenId].origin;
            require(originId != 0, "missing metadata");
            require(records[tokenId].children == 0, "subdomains not cleared");
            _pns.burn(tokenId);
```

```
            if (records[originId].children > 0) {
              records[originId].children -= 1;
            }
            records[tokenId].expire = 0;
            records[tokenId].capacity = 0;
            records[tokenId].origin = 0;
        }
    }
```

Listing 3.7: `PNSController::mintSubdomain()/burn()/burnBatch()`

If the privileged `root` account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol 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.

4 Conclusion

In this audit, we have analyzed the PNS design and implementation. The PNS is an open, decentralized domain name system on the Polkadot blockchain. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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: Numeric Errors. https://cwe.mitre.org/data/definitions/189.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.