SMART CONTRACT AUDIT REPORT

for

PANZ.NEXTGEN

Prepared By: Xiaomi Huang

PeckShield

March 14, 2023

Document Properties

Client	PANZ.NEXTGEN
Title	Smart Contract Audit Report
Target	PANZ.NEXTGEN
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 14, 2023	Xiaotao Wu	Final Release
1.0-rc	February 27, 2023	Xiaotao Wu	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the PANZ.NEXTGEN 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 PANZ.NEXTGEN

PANZ.NEXTGEN is a universe of next-degeneration products and IPs that embody the vision of establishing leading community-focused web3 creations. The protocol includes two parts: PANZ.FI and PANZ.PLAY. The first part is an NFT-backed loan protocol that offers easy access to instant liquidity for NFT holders of all project sizes. And the second part is a peer-to-peer protocol that allows users to host and join raffles for NFTs. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The PANZ.NEXTGEN

Item	Description
Name	PANZ.NEXTGEN
Website	https://panznextgen.com/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 14, 2023

In the following, we show the MD5 hash value of the related compressed file with the contracts for audit:

- MD5 (panznextgen-contract_20230223.zip) = 787aadfa5205639921b6ecb9e3485378

And this is the MD5 checksum value of the compressed file after fixes for the main issues found in the audit have been checked in:

- MD5 (panznextgen-contract_20230227.zip) = aef795c918df8e897192b59c5a9f77ed

Table 1.2: Deployed Contracts of The PANZ.NEXTGEN At The `Ethereum` Chain

Name	Contract Address
PANZ.PLAY	0xB3c8a4D882533bb072542d2F984B0B5ED91d246d
PANZ.FI	0xd60F44e24a7F13666911E61dB9Cd8E6E9ca769EA

1.2 About PeckShield

PeckShield Inc. [7] 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 1.3: 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 [6]:

- 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 1.3.

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 1.4.

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) [5], 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 1.5 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

Table 1.4: 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

Table 1.5: 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.

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 design and implementation of the PANZ.NEXTGEN 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 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	2	■■
Low	1	■
Informational	0	
Total	3	

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 2 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key PANZ.NEXTGEN Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Fee Withdraw Logic in PanzLending::withdrawFee()	Business Logic	Fixed
PVE-002	Low	Incorrect Sanity Checks in PanzLending::editOffer()	Business Logic	Fixed
PVE-003	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 for details.

3 | Detailed Results

3.1 Incorrect Fee Withdraw Logic in PanzLending::withdrawFee()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `PanzLending`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `PanzLending` contract provides an external `withdrawFee()` function for the privileged `owner` account to claim the platform fee from the contract. While reviewing its logic, we notice the current implementation is not correct.

To elaborate, we show below the related code snippet. It comes to our attention that the `feeBalance` is reset to 0 before the fee is sent to the `msg.sender` (line 76). Thus the fee claimed by the `msg.sender` is always 0 (line 77).

```
70     /**
71      * @dev claim platform fee
72      */
73     function withdrawFee() external onlyOwner nonReentrant {
74         if (feeBalance == 0) revert InsufficientBalance();
75 
76         feeBalance = 0;
77         (bool success,) = payable(msg.sender).call{value: feeBalance}("");
78         if(!success) revert PaymentFailed();
79     }
```

Listing 3.1: `PanzLending::withdrawFee()`

Recommendation Define a temporary variable to store the `feeBalance` before it is set to 0. An example revision is shown as follows:

```
/**
 * @dev claim platform fee
 */
function withdrawFee() external onlyOwner nonReentrant {
    uint256 fee = feeBalance;
    if (fee == 0) revert InsufficientBalance();

    feeBalance = 0;
    (bool success,) = payable(msg.sender).call{value: fee}("");
    if(!success) revert PaymentFailed();
}
```

Listing 3.2: `PanzLending::withdrawFee()`

Status This issue has been fixed.

3.2 Incorrect Sanity Checks in PanzLending::editOffer()

- ID: PVE-002
- Severity: Low
- Likelihood: High
- Impact: Low
- Target: `PanzLending`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `editOffer()` function of the `PanzLending` contract allows a user to edit an existing offer which was added by this user before. While reviewing its logic, we notice the current implementation fails to validate the given argument of `_count`.

To elaborate, we show below the code snippet of the `editOffer()` function. Since the sanity check for the input argument of `_count` requires `_count == 0 && _count > MAX_OFFER_COUNT`, which is always false (line 122). Thus the current check for `_count` is invalid.

```
118     /**
119      * @dev edit my offer
120      */
121     function editOffer(uint32 _offerId, uint32 _amount, uint8 _count) external payable
            nonReentrant {
122         if(_amount == 0   _count == 0 && _count > MAX_OFFER_COUNT) revert InvalidParams()
                ;
123         // check offer status
124         OfferData storage offerData = offers[_offerId];
125         if(offerData.count == 0) revert NoOfferFound();
126         if(offerData.owner != msg.sender) revert PermissionDenied();
127         if(offerData.count == _count && offerData.amount == _amount) revert
                InvalidParams();
```

```

        uint8 remainNum = offerData.remain;
        if (_count > offerData.count) {
            // add liquidity to pool
            offerData.remain += _count - offerData.count;
        } else if (_count < offerData.count) {
            // you cannot cut the liquidity to lower than borrowed number
            if(offerData.count - remainNum > _count) revert InvalidParams();
            offerData.remain -= offerData.count - _count;
        }

        // calculate amount
        uint256 newAmount = uint256(_amount) * PRICE_UNIT * (remainNum + _count - offerData.count);
        uint256 remainAmount = uint256(offerData.amount) * PRICE_UNIT * remainNum;
        if (newAmount > remainAmount) { // deposit
            uint256 totalAmount = newAmount - remainAmount;
            if(msg.value != totalAmount) revert InsufficientBalance();
        } else if (newAmount < remainAmount) { // withdraw
            uint256 totalAmount = remainAmount - newAmount;
            (bool success,) = payable(msg.sender).call{value: totalAmount}("");
            if(!success) revert PaymentFailed();
        }

        offerData.amount = _amount;
        offerData.count = _count;

        emit onOfferEdit(_offerId, msg.sender);
    }
```

Listing 3.3: PanzLending::editOffer()

Recommendation Add valid sanity checks for the above mentioned function.

Status This issue has been fixed.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `PanzLending/PanzRaffle`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `PANZ.NEXTGEN` protocol, there are two privileged accounts, i.e., `owner`, and `DEFAULT_ADMIN_ROLE`. These accounts play a critical role in governing and regulating the system-wide operations (e.g., set key parameters for the `PANZ.NEXTGEN` protocol, claim platform fee from the `PanzLending` contract, grant `MANAGER_ROLE` for the `PanzLending` contract, etc.). Our analysis shows that these privileged accounts need to be scrutinized. In the following, we use the `PanzLending` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
    /**
     * @dev set configurations
     */
    function setConfig(address _signer, uint16 _feeRate, uint16 _timeout) external
        onlyOwner {
        config.nonceTimeout = _timeout;
        config.feeRate = _feeRate;
        config.signer = _signer;
    }

    /**
     * @dev claim platform fee
     */
    function withdrawFee() external onlyOwner nonReentrant {
        if (feeBalance == 0) revert InsufficientBalance();

        feeBalance = 0;
        (bool success,) = payable(msg.sender).call{value: feeBalance}("");
        if(!success) revert PaymentFailed();
    }

    function grantRole(bytes32 role, address account) public virtual override onlyOwner
        {
        _grantRole(role, account);
    }
```

Listing 3.4: Example Privileged Operations in `PanzLending`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. 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.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes 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 PANZ.NEXTGEN design and implementation. PANZ.NEXTGEN is a universe of next-degeneration products and IPs that embody the vision of establishing leading community-focused web3 creations. The protocol includes two parts: PANZ.FI and PANZ.PLAY. The first part is an NFT-backed loan protocol that offers easy access to instant liquidity for NFT holders of all project sizes. And the second part is a peer-to-peer protocol that allows users to host and join raffles for NFTs. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

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

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

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

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

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

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