# SMART CONTRACT AUDIT REPORT

for

# InvtAI (Token Sale)

Prepared By: Xiaomi Huang

PeckShield

November 19, 2023

## Document Properties

| | |
|---|---|
| Client | InvtAI |
| Title | Smart Contract Audit Report |
| Target | InvtAI |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Colin Zhong, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | November 19, 2023 | Xuxian Jiang | Final Release |
| 1.0-rc1 | November 15, 2023 | Xuxian Jiang | 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 source code of the `InvtAI` token sale contract, 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 InvtAI

`InvtAI` aims to develop an `AI`-based blockchain consulting service that can help investors with their decisions. This `AI`-based service will utilize blockchain technology, news, and market updates to provide investors with an accurate and up-to-date overview of the blockchain market. Furthermore, the service will provide members with consulting services, ideas, and suggestions tailored to their individual requirements. This audit covers its token sale contract with the support of vesting schedules. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of The InvtAI

| Item | Description |
|---:|---|
| Name | InvtAI |
| Type | EVM Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | November 19, 2023 |

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

- https://github.com/InvtAIOfficial/tokensale-smartcontract.git (609edd)

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

- https://github.com/InvtAIOfficial/tokensale-smartcontract.git (02af11d)

## 1.2 About PeckShield

PeckShield Inc. [9] 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 |

**Likelihood**

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the 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) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

## 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 implementation of the `InvtAI` token sale contract. 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 | 2 | ■■ |
| Informational | 0 | |
| Total | 4 | |

We have so far identified a list of potential issues. 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 2 low-severity vulnerabilities.

Table 2.1: Key InvtAI Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Medium | Incorrect computeVestingScheduleIdForAddressAndPid() Logic | Business Logic | Resolved |
| PVE-002 | Low | Improved Validation in Sale Pool Addition | Coding Practices | Resolved |
| PVE-003 | Medium | Incorrect vestingStartTime Activation in harvestPool() | Business Logic | Resolved |
| PVE-004 | Low | Trust Issue of Admin Keys | Security Features | 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 Incorrect computeVestingScheduleIdForAddressAndPid() Logic

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `IvstSale`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

In the audited token sale contract, there is a helper routine `computeVestingScheduleIdForAddressAndPid()` that is designed to compute the vesting schedule identifier for a given address and a sale pool. Our analysis indicates that the generated identifier assumes the given address has no more than `2` vesting schedules, which can be relaxed.

In the following, we show the implementation of this specific routine. It has two arguments `_holder` and `_pid`. The logic is implemented to try the first possible identifier for the given `_holder`. If the pool id matches, we have successfully located the requested vesting schedule. Otherwise, it simply computes the next possible vesting schedule identifier. This computation may not be correct if the given `_holder` has more than `2` vesting schedules.

```
    function computeVestingScheduleIdForAddressAndPid(address _holder, uint256 _pid)
        external view returns (bytes32) {
        require(_pid < NUMBER_POOLS, "ComputeVestingScheduleId: Non valid pool id");
        bytes32 vestingScheduleId = computeVestingScheduleIdForAddressAndIndex(_holder,
            0);
        VestingSchedule memory vestingSchedule = vestingSchedules[vestingScheduleId];
        if (vestingSchedule.pid == _pid) {
            return vestingScheduleId;
        } else {
            return computeVestingScheduleIdForAddressAndIndex(_holder, 1);
        }
```

```
    }
```

Listing 3.1: `IvstSale::computeVestingScheduleIdForAddressAndPid()`

**Recommendation** Revise the above routine to accommodate the possibility of having multiple vesting schedules.

**Status** The issue has been fixed by this commit: `b26d171`.

## 3.2 Improved Validation in Sale Pool Addition

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `IvstSale`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The audited token sale contract is no exception. Specifically, if we examine the `IvstSale` contract, it has defined a number of pool-specific configuration, such as `_startTime` and `_endTime`. In the following, we show the corresponding routine that adds a new pool or updates an existing pool.

```
    function setPool(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _offeringAmountPool,
        uint256 _raisingAmountPool,
        uint256 _limitPerUserInLP,
        uint8 _pid,
        uint256 _vestingPercentage,
        uint256 _vestingCliff,
        uint256 _vestingDuration,
        uint256 _vestingSlicePeriodSeconds
    ) external onlyOwner {
        require(_pid < NUMBER_POOLS, "Operations: Pool does not exist");
        require(
            _vestingPercentage >= 0 && _vestingPercentage <= 100,
            "Operations: vesting percentage should exceeds 0 and interior 100"
        );
        require(_vestingDuration > 0, "duration must exceeds 0");
        require(_vestingSlicePeriodSeconds >= 1, "slicePeriodSeconds must be exceeds 1")
            ;
        require(_vestingSlicePeriodSeconds <= _vestingDuration, "slicePeriodSeconds must
             be interior duration");

```

```
        _poolInformation[_pid].startTime = _startTime;
        _poolInformation[_pid].endTime = _endTime;
        _poolInformation[_pid].offeringAmountPool = _offeringAmountPool;
        _poolInformation[_pid].raisingAmountPool = _raisingAmountPool;
        _poolInformation[_pid].limitPerUserInLP = _limitPerUserInLP;
        _poolInformation[_pid].vestingPercentage = _vestingPercentage;
        _poolInformation[_pid].vestingCliff = _vestingCliff;
        _poolInformation[_pid].vestingDuration = _vestingDuration;
        _poolInformation[_pid].vestingSlicePeriodSeconds = _vestingSlicePeriodSeconds;

        uint256 tokensDistributedAcrossPools;
        ...
    }
```

Listing 3.2: IvstSale :: setPool()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `_startTime` and `_endTime` may greatly affect the token sale.

Also, when a new payment token is added, there is a need to validate the given `decimals` is consistent with the token's decimals.

**Recommendation** Validate any changes regarding these pool-wide parameters to ensure they fall in an appropriate range.

**Status** The issue has been fixed by this commit: `b26d171`.

## 3.3 Incorrect vestingStartTime Activation in harvestPool()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `IvstSale`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

The token sale contract is designed with a global state `vestingStartTime`, which marks the vesting start time for everyone. However, this global start time needs to coordinate with all remaining pools as there is a need to ensure all pools have passed their `end times`.

To elaborate, we show below the related `harvestPool()` routine that initializes the vesting start time when the first user successfully claims the offering token. (Due to the vesting schedule, the claimed amount is only a portion of entire vesting amount.) However, it does not check whether all other pools have completed the sale. If there is a pool with uncompleted sale, the vesting for all users should not be started. To mitigate, we may have a pool-specific vesting start time to avoid the need to coordinate with other pools.

```
    function harvestPool(uint8 _pid) external nonReentrant notContract {
        require(harvestAllowed, "Harvest: Not allowed");
        // Checks whether it is too early to harvest
        require(block.timestamp > _poolInformation[_pid].endTime, "Harvest: Too early");

        // Checks whether pool id is valid
        require(_pid < NUMBER_POOLS, "Harvest: Non valid pool id");

        // Checks whether the user has participated
        require(_userInfo[msg.sender][_pid].amountPool > 0, "Harvest: Did not
            participate");

        // Checks whether the user has already harvested
        require(!_userInfo[msg.sender][_pid].claimedPool, "Harvest: Already done");

        // Updates the harvest status
        _userInfo[msg.sender][_pid].claimedPool = true;

        // Updates the vesting startTime
        if (vestingStartTime == 0) {
            vestingStartTime = block.timestamp;
        }

        uint256 offeringTokenAmount = _calculateOfferingAmountPool(msg.sender, _pid);
        ...
    }
```

Listing 3.3: `IvstSale::harvestPool()`

**Recommendation** Ensure all pools have completed the sale if the vesting start time is initialized.

**Status** This issue has been fixed by having a pool-specific vesting start time.

## 3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

### Description

In InvtAI, there is a privileged administrative account, i.e., owner. The administrative account plays a critical role in governing and regulating the token sale operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the IvstSale contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function finalWithdraw(address[] calldata _tokens, uint256 _offerAmount) external onlyOwner {
        if (_offerAmount > 0) {
            offeringToken.safeTransfer(msg.sender, _offerAmount);
        }

        uint256 ethBalance = address(this).balance;
        payable(msg.sender).transfer(ethBalance);

        uint256[] memory _amounts = new uint256[](_tokens.length);
        for (uint256 i = 0; i < _tokens.length; i++) {
            _amounts[i] = IERC20(_tokens[i]).balanceOf(address(this));
            if (_amounts[i] > 0) {
                IERC20(_tokens[i]).safeTransfer(msg.sender, _amounts[i]);
            }
        }

        emit AdminWithdraw(_offerAmount, ethBalance, _tokens, _amounts);
    }

    function recoverWrongTokens(address _tokenAddress, uint256 _tokenAmount) external onlyOwner {
        require(!isPaymentToken[_tokenAddress] && !isStableToken[_tokenAddress], "Recover: Cannot be payment token");
        require(_tokenAddress != address(offeringToken), "Recover: Cannot be offering token");

        IERC20(_tokenAddress).safeTransfer(msg.sender, _tokenAmount);

        emit AdminTokenRecovery(_tokenAddress, _tokenAmount);
    }

    function setOfferingToken(address _tokenAddress) external onlyOwner {
```

```
        require(_tokenAddress != address(0), "OfferingToken: Zero address");

        offeringToken = IERC20(_tokenAddress);

        emit OfferingTokenSet(_tokenAddress);
    }

    function setPool(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _offeringAmountPool,
        uint256 _raisingAmountPool,
        uint256 _limitPerUserInLP,
        uint8 _pid,
        uint256 _vestingPercentage,
        uint256 _vestingCliff,
        uint256 _vestingDuration,
        uint256 _vestingSlicePeriodSeconds
    ) external onlyOwner {
        require(_pid < NUMBER_POOLS, "Operations: Pool does not exist");
        ...
    }
```

Listing 3.4: Example Privileged Operations in `IvstSale`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative 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 resolved as the admin will be prevented from changing the offering token after the sale ends.

# 4 Conclusion

In this audit, we have analyzed the design and implementation of the InvtAI protocol, which aims to develop an AI-based blockchain consulting service that can help investors with their decisions. This AI-based service will utilize blockchain technology, news, and market updates to provide investors with an accurate and up-to-date overview of the blockchain market. Furthermore, the service will provide members with consulting services, ideas, and suggestions tailored to their individual requirements. This audit covers its token sale contract with vesting schedules. 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-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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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

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