SECURITY AUDIT REPORT

for

InteNet Protocol

Prepared By: Xiaomi Huang

PeckShield

January 17, 2025

Document Properties

Client	InteNet Protocol
Title	Security Audit Report
Target	InteNet
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 17, 2025	Xuxian Jiang	Final Release
1.0-rc	January 16, 2025	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the InteNet 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 InteNet

InteNet protocol aims to build an innovative DeFi platform designed to cater to the launch, liquidity management, and automated market operations of AI agents and meme assets. With features such as a Launch Pad, a Native DEX, and AI Agent Modules, InteNet offers a seamless experience for asset issuance and trading. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of InteNet

Item	Description
Name	InteNet Protocol
Type	Solidity
Language	EVM
Audit Method	Whitebox
Latest Audit Report	January 17, 2025

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the InteNet protocol by design keeps track of the virtual reserves (not actual liquidity of paired asset) when the bonding-managed internal pair is created.

- https://github.com/JarvisAgentLab/probable-system.git (04c3634)

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

- https://github.com/JarvisAgentLab/probable-system.git (0da3e99)

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 1.2: 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 [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 accordingly classified into four categories, 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) [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 1.4 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 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 `InteNet` implementations. 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 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 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved buy()/sell() Logic in INTRouterLibrary	Business Logic	Resolved
PVE-002	Low	Possible MarketCap Manipulation in Bonding	Time And State	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Improved buy()/sell() Logic in INTRouterLibrary

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

Description

The `InteNet` protocol has a key library named `INTRouterLibrary` that contains the sell/buy logic for the launched tokens. In the process of examining the related token sell/buy implementation, we notice it can be improved to explicitly indicate the fund source for the associated transaction.

In the following, we show the implementation of the example `buy()` routine. As the name indicates, this routine is used to buy the new token being launched with the given `assetToken`. And the `assetToken` amount is provided by the calling user, i.e., `msg.sender`, not the given input argument of `to`[1]. With that, we suggest to use `msg.sender` when transferring the `assetToken` amount to the pair (line 245) and transferring the buy fee to the treasury (line 247). Note the `sell()` counterpart routine can be similarly improved.

```
234     function buy(
235         INTFactory factory,
236         address assetToken,
237         uint256 amountIn,
238         address tokenAddress,
239         address to
240     ) public returns (uint256, uint256) {
241         if (tokenAddress == address(0)) revert TokenIsZeroAddress();
242         if (to == address(0)) revert RecipientIsZeroAddress();
243         if (amountIn == 0) revert InputAmountIsZero();
```

[1] Fortunately, the current implementation hardcodes the to state with `msg.sender`. Nevertheless, at least semantically, the funding source is `msg.sender`, not to.

```
        address pair = factory.getPair(tokenAddress, assetToken);

        (uint256 amountOut, uint256 txFee) = quoteBuy(
            factory,
            assetToken,
            tokenAddress,
            amountIn
        );
        uint256 amount = amountIn - txFee;

        IERC20(assetToken).safeTransferFrom(to, pair, amount);

        collectFee(factory, assetToken, to, address(0), tokenAddress, txFee);

        IINTPair(pair).transferTo(to, amountOut);

        IINTPair(pair).swap(0, amountOut, amount, 0);

        (uint256 reserveA, uint256 reserveB) = IINTPair(pair).getReserves();

        emit Buy(
            to,
            tokenAddress,
            amountOut,
            amount,
            txFee,
            reserveA,
            reserveB,
            block.timestamp
        );

        return (amountIn, amountOut);
    }
```

Listing 3.1: `INTRouterLibrary::buy()`

Moreover, the `sell()` routine has another issue in updating the pair reserves. In particular, they are currently updated as `pair.swap(amountIn, 0, 0, amountOut)` (line 206), which should be revised as `pair.swap(amountIn, 0, 0, amountOut + txFee)`

Recommendation Revise the above-mentioned `buy()/sell()` routines to properly use the funding source for the associated token buy/sell transactions and update the pair reserves.

Status This issue has been resolved in the following commit: `0da3e99`.

3.2 Possible MarketCap Manipulation in Bonding

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

Description

The `InteNet` protocol has a unique boost mechanism that enables the expedite token launch and liquidity provision. One key factor behind the boost mechanism is the current market cap of the token being launched. While examining the logic to determine current token market cap, we notice the calculation method may be manipulated.

In the following, we show the implementation of the related `calculateMarketCap()` routine. As the name indicates, it calculates the market cap of a token by using the configured oracle price and liquidity reserves. While the oracle price makes use of the time-weighted average price, the liquidity reserves are based on the instance reserves from the trading pair. As a result, the liquidity reserves can be manipulated with a flashloan and the boost mechanism can therefore be affected.

```
    function calculateMarketCap(address token) public returns (uint256) {
        Token storage _token = tokenInfo[token];
        TokenStatus status = _token.status;
        if (status == TokenStatus.None) revert InvalidToken();

        uint256 assetPrice = IOracle(oracle).getAssetPrice();
        if (assetPrice == 0) revert InvalidAssetPrice();

        uint256 tokenReserve;
        uint256 assetReserve;
        address pair = _token.pair;

        if (status == TokenStatus.BondingCurve) {
            (tokenReserve, assetReserve) = IINTPair(pair).getReserves();
        } else {
            (uint256 reserve0, uint256 reserve1, ) = IUniswapV2Pair(pair)
                .getReserves();
            bool isToken0 = token < assetToken;

            (tokenReserve, assetReserve) = isToken0
                ? (reserve0, reserve1)
                : (reserve1, reserve0);
        }

        if (tokenReserve == 0  assetReserve == 0) revert InvalidReserves();

```

```
        // Get total supply
        uint256 totalSupply = IERC20(token).totalSupply();

        // Calculate market cap: totalSupply * tokenPrice * assetPrice / 1e18
        uint256 marketCap = (assetPrice * totalSupply * assetReserve) /
            tokenReserve /
            1e18;

        return marketCap;
    }
```

Listing 3.2: `Bonding::calculateMarketCap()`

Recommendation Revise the above routine to reliably compute the market cap.

Status This issue has been resolved as it may only occur when a token is graduated and is also greatly mitigated with the presence of buy/sell tax.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the audited contracts, there is a privileged `owner` account (with the `DEFAULT_ADMIN_ROLE` role). This account plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, manage delegates, execute privileged operations, upgrade contracts, etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Bonding` contract as an example and show the representative functions potentially affected by the privileged account.

```
    function setInitialSupply(uint256 newSupply) public onlyOwner {
        initialSupply = newSupply;

        emit InitializeSupplySet(newSupply);
    }
    ...
    function setFee(uint256 newFee, address newFeeTo) public onlyOwner {
        fee = newFee;
        feeTo = newFeeTo;

        emit FeeSet(newFee, newFeeTo);
```

```
    }
    ...
    function setOracle(address newOracle) public onlyOwner {
        if (IOracle(newOracle).getAssetPrice() == 0) revert InvalidOracle();

        oracle = IOracle(newOracle);

        emit OracleSet(newOracle);
    }
    ...
    function setInitialMarketCap(uint256 newMarketCap) public onlyOwner {
        if (newMarketCap == 0) revert InvalidMarketCap();

        initialMarketCap = newMarketCap;

        emit InitialMarketCapSet(newMarketCap);
    }
    ...
    function setGradMarketCap(uint256 newMarketCap) public onlyOwner {
        if (newMarketCap < initialMarketCap) revert InvalidMarketCap();

        gradMarketCap = newMarketCap;

        emit GradMarketCapSet(newMarketCap);
    }
    ...
    function setLockedTime(uint256 newLockedTime) public onlyOwner {
        if (newLockedTime <= 365 days) revert InvalidLockTime();
        lockedTime = newLockedTime;

        emit LockedTimeSet(newLockedTime);
    }
    ...
    function setFactory(address newFactory) public onlyOwner {
        factory = INTFactory(newFactory);

        emit FactorySet(newFactory);
    }
    ...
    function setTokenFactory(address newTokenFactory) public onlyOwner {
        tokenFactory = INTERC20Factory(newTokenFactory);

        emit TokenFactorySet(newTokenFactory);
    }
    ...
    function setLockFactory(address newLockFactory) public onlyOwner {
        lockFactory = LockFactory(newLockFactory);

        emit LockFactorySet(newLockFactory);
    }
    ...
    function setExtRouter(address newExtRouter) public onlyOwner {
```

```
        extRouter = IExtRouter(newExtRouter);

        emit ExtRouterSet(newExtRouter);
    }
    ...
    function setAssetToken(address newAssetToken) public onlyOwner {
        assetToken = newAssetToken;

        emit AssetTokenSet(newAssetToken);
    }
```

Listing 3.3: Privileged Operations in `Bonding`

We understand the need of the privileged functions for proper protocol operations, but at the same time the extra power to the privileged admin may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

In the meantime, a number of protocol contracts make use of the proxy contract to allow for future upgrades. The upgrade is a privileged operation and the management of the related admin key also falls in this trust issue.

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 mitigated as the team plans to assign admin role to a multi-sig wallet.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `InteNet` protocol, which is an innovative `DeFi` platform designed to cater to the launch, liquidity management, and automated market operations of `AI agents` and `meme assets`. With features such as a `Launch Pad`, a `Native DEX`, and `AI Agent Modules`, `InteNet` offers a seamless experience for asset issuance and trading. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-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.