ExVul

SMART CONTRACT AUDIT REPORT

Janction Smart Contract

OCTOBER 2025

Contents

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **Janction** to review smart contract implementation. The assessment was conducted in accordance with our systematic approach to evaluate potential security issues based upon customer requirement. The report provides detailed recommendations to resolve the issue and provide additional suggestions or recommendations for improvement.

The outcome of the assessment outlined in chapter 3 provides the system's owners a full description of the vulnerabilities identified, the associated risk rating for each vulnerability, and detailed recommendations that will resolve the underlying technical issue.

1.1 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10] which is the gold standard in risk assessment using the following risk models:

- **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**: determine the overall criticality of the risk.

Likelihood can be: High, Medium and Low and impact are categorized into: High, Medium, Low, Informational. Severity is determined by likelihood and impact and can be classified into five categories accordingly: Critical, High, Medium, Low, Informational shown in table 1.1.

Likelihood \ IMPACT	Informational	Low	Medium	High
High	INFO	MEDIUM	HIGH	CRITICAL
Medium	INFO	LOW	MEDIUM	HIGH
Low	INFO	LOW	LOW	MEDIUM

Table 1.1 Overall Risk Severity

To evaluate the risk, we will be going through a list of items, and each would be labelled with a severity category. The audit was performed with a systematic approach guided by a comprehensive assessment list carefully designed to identify known and impactful security issues. If our tool or analysis does not identify any issue, the contract can be considered safe regarding the assessed item. For any discovered issue, we might further deploy contracts on our private test environment 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 1.2.

- **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.
- **Code and business security testing**: 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.

Category	Assessment Item
Basic Coding Assessment	• Apply Verification Control • Authorization Access Control • Forged Transfer Vulnerability • Forged Transfer Notification • Numeric Overflow • Transaction Rollback Attack • Transaction Block Stuffing Attack • Soft Fail Attack • Hard Fail Attack • Abnormal Memo • Abnormal Resource Consumption • Secure Random Number

Advanced Source Code Scrutiny	• Asset Security • Cryptography Security • Business Logic Review • Source Code Functional Verification • Account Authorization Control • Sensitive Information Disclosure • Circuit Breaker • Blacklist Control • System API Call Analysis • Contract Deployment Consistency Check • Abnormal Resource Consumption
Additional Recommendations	• Semantic Consistency Checks • Following Other Best Practices

Table 1.2: The Full List of Assessment Items

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [14], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development.

2. FINDINGS OVERVIEW

2.1 Project Info And Contract Address

Project Name	Audit Time	Language
Janction	30/09/2025 - 07/10/2025	Solidity

Repository
https://github.com/Janction-R-D/janction-contracts

Commit Hash
b5f141af2619488b2e281051e10e2f5f1be35b8b

2.2 Summary

Severity	Found
CRITICAL	0
HIGH	0
MEDIUM	3
LOW	5
INFO	2

2.3 Key Findings

Severity	Findings Title	Status
MEDIUM	Incorrect Refund Calculation in Payment Plan Termination	Fixed
MEDIUM	Owner can withdraw escrowed NFTs breaking user escrow records	Acknowledge
MEDIUM	Owner can centralize arbitrarily increase maximum supply	Acknowledge
LOW	Signature Threshold Bypass when Payer Equals Recipient	Acknowledge
LOW	Administrator can bypass multi signature threshold	Acknowledge
LOW	Refund goes to wrong address when msg.sender != payer	Acknowledge
LOW	Signature Malleability Attack	Fixed
LOW	Zero amount distribution allows successful execution	Acknowledge
INFO	Implementation contract can be initialized and upgraded by anyone	Fixed
INFO	Unnecessary zero amount transfer to treasury	Fixed

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 Incorrect Refund Calculation in Payment Plan Termination

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`PaymentImpl.sol`

DESCRIPTION:

The `stopPaymentPlan` and `stopPaymentPlanAdmin` functions calculate refund amounts incorrectly when a payment plan has been fully executed. The refund calculation uses `remainingBalance = plan.totalAmount - plan.paidHours * plan.hourlyAmount`, which can result in non-zero values even when all payments have been completed.

```
function stopPaymentPlan(
    bytes32 paymentId,
    EIP712Signature[] calldata signatures
) external {
    //...

    try this.releaseHourlyPayment(paymentId) {} catch {}

    uint256 remainingBalance = plan.totalAmount -
        plan.paidHours *
        plan.hourlyAmount;

    IERC20(plan.currency).safeTransfer(plan.payer, remainingBalance);

    plan.stopped = true;

    emit PaymentPlanStopped(
        paymentId,
        plan.payer,
        plan.recipient,
        remainingBalance
    );
}
```

```
function stopPaymentPlanAdmin(bytes32 paymentId) external onlyAdmin {
    //...

    // Try to release any due hourly payment before stopping
    try this.releaseHourlyPayment(paymentId) {} catch {}

    uint256 remainingBalance = plan.totalAmount -
        plan.paidHours *
        plan.hourlyAmount;

    IERC20(plan.currency).safeTransfer(plan.payer, remainingBalance);

    plan.stopped = true;

    emit PaymentPlanStopped(
        paymentId,
        plan.payer,
        plan.recipient,
        remainingBalance
    );
}

function releaseHourlyPayment(bytes32 paymentId) external {
    //...

    // Determine the actual hours to pay without exceeding totalHours
    uint256 hoursToPay = (plan.paidHours + passedHours > plan.totalHours)
        ? plan.totalHours - plan.paidHours
        : passedHours;

    // If this is the final payment, transfer the remaining balance to
   avoid overpayment
    uint256 amountToTransfer = (plan.paidHours + hoursToPay ==
        plan.totalHours)
        ? plan.totalAmount - plan.paidHours * plan.hourlyAmount
        : plan.hourlyAmount * hoursToPay;
}
```

Example A - Normal (Evenly Divisible):

- `createPaymentPlan` with `totalAmount = 100`, `totalHours = 10`, `hourlyAmount`

`= 10`

- After 9 payments: `paidHours = 9`, released amount = 90, remaining balance = 10
- When the final payment, `releaseHourlyPayment` releases the last 10 tokens, now all 100 tokens are distributed
- Stop function calculates `remainingBalance = plan.totalAmount(100) - plan.paidHours(10) * plan.hourlyAmount(10) = 0`
- `safeTransfer` succeeds with 0 amount (no refund needed)

Example B - Problematic (Not Evenly Divisible):

- `createPaymentPlan` with `totalAmount = 98`, `totalHours = 10`, `hourlyAmount = 9`
- After 9 payments: `paidHours = 9`, released amount = 81, remaining balance = 17
- When the final payment, `releaseHourlyPayment` releases the remaining 17 tokens, now all 98 tokens are distributed
- Stop function calculates `remainingBalance = plan.totalAmount(98) - plan.paidHours(10) * plan.hourlyAmount(9) = 8`
- `safeTransfer` fails because contract tries to refund 8 tokens that were already paid to recipient

Impact:

Completed payment plans cannot be properly terminated due to incorrect refund calculations attempting to transfer already-distributed funds.

RECOMMENDATIONS:

Modify `stopPaymentPlan` and `stopPaymentPlanAdmin` to check if the payment plan is fully completed in the refund calculation.

```
    try this.releaseHourlyPayment(paymentId) {} catch {}

-   uint256 remainingBalance = plan.totalAmount -
-       plan.paidHours *
-       plan.hourlyAmount;

-   IERC20(plan.currency).safeTransfer(plan.payer, remainingBalance);

+   uint256 remainingBalance = 0;
+   if (plan.paidHours < plan.totalHours) {
+       remainingBalance = plan.totalAmount - plan.paidHours *
    plan.hourlyAmount;
```

```
+    }
+
+    if (remainingBalance > 0) {
+        IERC20(plan.currency).safeTransfer(plan.payer, remainingBalance);
+    }
```

3.2 Owner can withdraw escrowed NFTs breaking user escrow records

SEVERITY: MEDIUM **STATUS:** Acknowledge

PATH:

`NFTEscrowImpl.sol`

DESCRIPTION:

The centralized `withdraw` function allows the owner to transfer any NFT without checking the `ownerByToken` mapping. Withdrawal of user-escrowed NFTs breaks the escrow system.

```
function withdraw(uint256[] memory tokenIdList, address to) public
    onlyOwner {
     for(uint256 i = 0; i < tokenIdList.length; ++i) {
         IERC721(janctionNFT).transferFrom(address(this), to,
    tokenIdList[i]);
     }
}

function escrow(uint256 tokenId) public {
     ownerByToken[tokenId] = msg.sender;
     IERC721(janctionNFT).transferFrom(msg.sender, address(this), tokenId);
     emit Escrowed(msg.sender, tokenId);
}

function unescrow(uint256 tokenId) public {
     require(ownerByToken[tokenId] == msg.sender, ”not token id owner”);
     delete ownerByToken[tokenId];
     IERC721(janctionNFT).transferFrom(address(this), msg.sender, tokenId);
     emit Unescrowed(msg.sender, tokenId);
}
```

RECOMMENDATIONS:

Add a check to prevent withdrawal of escrowed NFTs owned by users.

```
function withdraw(uint256[] memory tokenIdList, address to) public
    onlyOwner {
     for(uint256 i = 0; i < tokenIdList.length; ++i) {
+        require(ownerByToken[tokenIdList[i]] == address(0), ”cannot
    withdraw escrowed NFT”);
         IERC721(janctionNFT).transferFrom(address(this), to,
    tokenIdList[i]);
     }
}
```

3.3 Owner can centralize arbitrarily increase maximum supply

SEVERITY: MEDIUM **STATUS:** Acknowledge

PATH:

`JanctionNFT.sol`

DESCRIPTION:

The owner can increase the maximum supply at any time through the `setMaxSupply` function, as long as the new value is greater than the current total supply.

```
function setMaxSupply(uint256 maxSupply) public onlyOwner {
    require(maxSupply > totalSupply(), "invalid max supply");
    MAX_SUPPLY = maxSupply;
}
```

RECOMMENDATIONS:

Restrict the `setMaxSupply` function to only allow decreasing the maximum supply, preventing unlimited inflation while still allowing conservative supply adjustments.

```
function setMaxSupply(uint256 maxSupply) public onlyOwner {
    require(maxSupply > totalSupply(), "invalid max supply");
+   require(maxSupply < MAX_SUPPLY, "cannot increase max supply");
    MAX_SUPPLY = maxSupply;
}
```

3.4 Signature Threshold Bypass when Payer Equals Recipient

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

`PaymentImpl.sol`

DESCRIPTION:

The `stopPaymentPlan` function allows signature threshold bypass when payer and recipient are the same address. When payer equals recipient, the same address appears twice in the signers array, allowing a single signature to be counted twice, bypassing the multi-signature requirement.

```
function stopPaymentPlan(
    bytes32 paymentId,
    EIP712Signature[] calldata signatures
) external {
    // ...
    address[] memory signers = new address[](3);
    signers[0] = plan.payer;
    signers[1] = plan.recipient;
    signers[2] = _administrator;

    uint256 validSignatures = 0;
    for (uint256 i = 0; i < signatures.length; i++) {
        address recoveredAddr = _recoverEIP712Signer(
            _hashTypedDataV4(
                keccak256(
                    abi.encode(
                        STOP_PAYMENT_PLAN_TYPEHASH,
                        paymentId,
                        signatures[i].deadline
                    )
                )
            ),
            signatures[i]
        );

        require(
```

```
            recoveredAddr == signatures[i].signer,
            "recovered address not equal to signer"
        );

        for (uint256 j = 0; j < signers.length; j++) {
            if (recoveredAddr == signers[j]) {
                validSignatures++;
                signers[j] = address(0); // Prevent double-counting
                break;
            }
        }
    }

    require(
        validSignatures >= signatureThreshold,
        "insufficient valid signatures"
    );
    // ...
}
```

Example:

- Set `signatureThreshold = 2`
- Create payment plan with `payer == recipient` (same address)
- Submit one signature from the payer/recipient
- The signature matches both `signers[0]` and `signers[1]` (same address)
- `validSignatures` becomes 2, satisfying the threshold requirement with just one signature

RECOMMENDATIONS:

Prevent payer and recipient from being the same address.

```
function createPaymentPlan(
    address payer,
    address recipient,
    address currency,
    uint256 totalAmount,
    uint256 totalHours,
    bytes32 data
) external {
    require(isCurrencyWhitelisted[currency], "currency not whitelisted");
    require(
```

```
        block.timestamp - lastPurchaseTime[data] >= purchaseInterval,
        "item recently purchased"
    );
+   require(payer != recipient, "payer and recipient cannot be the same");
```

3.5 Administrator can bypass multi signature threshold

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

`PaymentImpl.sol`

DESCRIPTION:

The administrator is included as one of the three signers in the multi-signature verification process, but also has a separate `stopPaymentPlanAdmin` function that allows bypassing the signature threshold requirement.

```
function stopPaymentPlan(
    bytes32 paymentId,
    EIP712Signature[] calldata signatures
) external {
    require(
        signatures.length >= signatureThreshold,
        "insufficient signatures"
    );
    // ...
    address[] memory signers = new address[](3);
    signers[0] = plan.payer;
    signers[1] = plan.recipient;
    signers[2] = _administrator;
}

function stopPaymentPlanAdmin(bytes32 paymentId) external onlyAdmin {
    // ...
}
```

RECOMMENDATIONS:

Either remove the administrator bypass function or implement additional restrictions. Or remove administrator from multi-signature participants and require true consensus between payer and recipient only.

```
- function stopPaymentPlanAdmin(bytes32 paymentId) external onlyAdmin {
+ function emergencyStopPaymentPlan(bytes32 paymentId) external onlyAdmin
    {
+       require(paused, "emergency stop only when paused");
+       // ...
}
```

3.6 Refund goes to wrong address when msg.sender != payer

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

`PaymentImpl.sol`

DESCRIPTION:

Tokens are transferred from `msg.sender` but the payment plan records the `payer` parameter. When the payment plan is stopped, refunds are sent to `plan.payer` instead of the actual token sender. Also, the real payer is `msg.sender` but not the parameter `payer`.

```
function createPaymentPlan(
    address payer,
    address recipient,
    address currency,
    uint256 totalAmount,
    uint256 totalHours,
    bytes32 data
) external {
    IERC20(currency).safeTransferFrom(
        msg.sender,
        address(this),
        totalAmount
    );

    _paymentPlans[paymentId] = PaymentPlan({
        payer: payer,
        // ...
    });
}

function stopPaymentPlan(...) external {
    IERC20(plan.currency).safeTransfer(
        plan.payer,
        remainingBalance
    );
}
```

RECOMMENDATIONS:

Add validation to ensure caller is the payer.

```
function createPaymentPlan(
    address payer,
    address recipient,
    address currency,
    uint256 totalAmount,
    uint256 totalHours,
    bytes32 data
) external {
+   require(msg.sender == payer, "Caller must be payer");
    require(isCurrencyWhitelisted[currency], "currency not whitelisted");
    // ... rest of code
}
```

3.7 Signature Malleability Attack

SEVERITY: LOW **STATUS:** Fixed

PATH:

`JasmyRewards.sol`

DESCRIPTION:

Using native `ecrecover` without checking signature canonicality allows attackers to create malleable signatures by modifying the `s` and `v` values to bypass signature verification.

```
function _recoverEIP712Signer(
    bytes32 digest,
    EIP712Signature memory signature
) internal view returns (address) {
    require(block.timestamp < signature.deadline, "signature expired");
    address recoveredAddress = ecrecover(
        digest,
        signature.v,
        signature.r,
        signature.s
    );
    return recoveredAddress;
}
```

Additionally, `ECDSA.recover` is better suited for `_hashTypedDataV4` and follows EIP712 best practices.

```
/**
 * @dev Given an already
    https://eips.ethereum.org/EIPS/eip-712#definition-of-hashstruct[hashed
    struct], this
 * function returns the hash of the fully encoded EIP712 message for this
    domain.
 *
 * This hash can be used together with {ECDSA-recover} to obtain the
    signer of a message. For example:
 *
```

```
 * ```solidity
 * bytes32 digest = _hashTypedDataV4(keccak256(abi.encode(
 *     keccak256("Mail(address to,string contents)"),
 *     mailTo,
 *     keccak256(bytes(mailContents))
 * )));
 * address signer = ECDSA.recover(digest, signature);
 * ```
 */
function _hashTypedDataV4(bytes32 structHash) internal view virtual
   returns (bytes32) {
    return MessageHashUtils.toTypedDataHash(_domainSeparatorV4(),
   structHash);
}
```

RECOMMENDATIONS:

Use OpenZeppelin's ECDSA.recover() which includes malleability protection.

```
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";

function _recoverEIP712Signer(
    bytes32 digest,
    EIP712Signature memory signature
) internal view returns (address) {
    require(block.timestamp < signature.deadline, "signature expired");
-   address recoveredAddress = ecrecover(
-       digest,
-       signature.v,
-       signature.r,
-       signature.s
-   );
+   address recoveredAddress = ECDSA.recover(digest, signature);
    return recoveredAddress;
}
```

3.8 Zero amount distribution allows successful execution

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

`Distribution.sol`

DESCRIPTION:

The `distribute` function lacks input validation for zero amounts, enabling successful execution when `totalAmount` is set to 0 with empty `beneficiaries` and `rewards` arrays.

```
function distribute(
    address currency,
    uint256 totalAmount,
    address[] memory beneficiaries,
    uint256[] memory rewards
) external {
    require(
        beneficiaries.length == rewards.length,
        ”array length not equal”
    );
    require(isCurrencyWhitelisted[currency], ”currency not whitelisted”);

    uint256 remainingAmount = totalAmount;
    for (uint256 i = 0; i < beneficiaries.length; ++i) {
        IERC20(currency).safeTransferFrom(
            msg.sender,
            beneficiaries[i],
            rewards[i]
        );
        remainingAmount -= rewards[i];
    }

    IERC20(currency).safeTransferFrom(
        msg.sender,
        treasury,
        remainingAmount
    );
```

```
    emit Distributed(
        msg.sender,
        treasury,
        currency,
        totalAmount,
        remainingAmount,
        beneficiaries,
        rewards
    );
}
```

RECOMMENDATIONS:

Add input validation to prevent zero amount distributions and empty arrays.

```
function distribute(
    address currency,
    uint256 totalAmount,
    address[] memory beneficiaries,
    uint256[] memory rewards
) external {
+   require(totalAmount > 0, "total amount must be greater than zero");
+   require(beneficiaries.length > 0, "beneficiaries must be greater than
    zero");
    require(
        beneficiaries.length == rewards.length,
        "array length not equal"
    );
    require(isCurrencyWhitelisted[currency], "currency not whitelisted");
```

3.9 Implementation contract can be initialized and upgraded by anyone

SEVERITY: INFO **STATUS:** Fixed

PATH:

`NFTEscrowImpl.sol`

DESCRIPTION:

The contract lacks `_disableInitializers()` in the constructor. Anyone can call `initialize` on the implementation contract address itself and become its owner, then perform UUPS upgrades on the implementation contract.

```
contract NFTEscrowImpl is UUPSUpgradeable, OwnableUpgradeable {
    function initialize(address initialOwner, address nft) public
    initializer {
        __Ownable_init(initialOwner);
        janctionNFT = nft;
    }

    function _authorizeUpgrade(
        address newImplementation
    ) internal onlyOwner virtual override {}
}
```

RECOMMENDATIONS:

Consider calling `_disableInitializers()` in the constructor.

```
contract NFTEscrowImpl is UUPSUpgradeable, OwnableUpgradeable {
+   constructor() {
+       _disableInitializers();
+   }
+
    function initialize(address initialOwner, address nft) public
    initializer {
        __Ownable_init(initialOwner);
        janctionNFT = nft;
```

```
    }
```

3.10 Unnecessary zero amount transfer to treasury

SEVERITY: INFO **STATUS:** Fixed

PATH:

`Distribution.sol`

DESCRIPTION:

The `distribute` function always transfers `remainingAmount` to the treasury, causing unnecessary gas consumption when `remainingAmount` is 0.

```
function distribute(
    address currency,
    uint256 totalAmount,
    address[] memory beneficiaries,
    uint256[] memory rewards
) external {
    require(
        beneficiaries.length == rewards.length,
        "array length not equal"
    );
    require(isCurrencyWhitelisted[currency], "currency not whitelisted");

    uint256 remainingAmount = totalAmount;
    for (uint256 i = 0; i < beneficiaries.length; ++i) {
        IERC20(currency).safeTransferFrom(
            msg.sender,
            beneficiaries[i],
            rewards[i]
        );
        remainingAmount -= rewards[i];
    }

    IERC20(currency).safeTransferFrom(
        msg.sender,
        treasury,
        remainingAmount
    );
```

```
    emit Distributed(
        msg.sender,
        treasury,
        currency,
        totalAmount,
        remainingAmount,
        beneficiaries,
        rewards
    );
}
```

RECOMMENDATIONS:

Add a condition to only transfer when `remainingAmount` is greater than 0.

```
    for (uint256 i = 0; i < beneficiaries.length; ++i) {
        IERC20(currency).safeTransferFrom(
            msg.sender,
            beneficiaries[i],
            rewards[i]
        );
        remainingAmount -= rewards[i];
    }

-   IERC20(currency).safeTransferFrom(
-       msg.sender,
-       treasury,
-       remainingAmount
-   );
+   if (remainingAmount > 0) {
+       IERC20(currency).safeTransferFrom(
+           msg.sender,
+           treasury,
+           remainingAmount
+       );
+   }
```

4. CONCLUSION

In this audit, we thoroughly analyzed **Janction** smart contract implementation. The problems found are described and explained in detail in Section 3. The problems found in the audit have been communicated to the project leader. We therefore consider the audit result to be **PASSED**.

To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5. APPENDIX

5.1 Basic Coding Assessment

5.1.1 Apply Verification Control

Description	The security of apply verification
Result	Not found
Severity	CRITICAL

5.1.2 Authorization Access Control

Description	Permission checks for external integral functions
Result	Not found
Severity	CRITICAL

5.1.3 Forged Transfer Vulnerability

Description	Assess whether there is a forged transfer notification vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.4 Transaction Rollback Attack

Description	Assess whether there is transaction rollback attack vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.5 Transaction Block Stuffing Attack

Description	Assess whether there is transaction blocking attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.6 Soft Fail Attack Assessment

Description	Assess whether there is soft fail attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.7 Hard Fail Attack Assessment

Description	Examine for hard fail attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.8 Abnormal Memo Assessment

Description	Assess whether there is abnormal memo vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.9 Abnormal Resource Consumption

Description	Examine whether abnormal resource consumption in contract processing
Result	Not found
Severity	CRITICAL

5.1.10 Random Number Security

Description	Examine whether the code uses insecure random number
Result	Not found
Severity	CRITICAL

5.2 Advanced Code Scrutiny

5.2.1 Cryptography Security

Description	Examine for weakness in cryptograph implementation
Result	Not found
Severity	HIGH

5.2.2 Account Permission Control

Description	Examine permission control issue in the contract
Result	Not found
Severity	MEDIUM

5.2.3 Malicious Code Behavior

Description	Examine whether sensitive behavior present in the code
Result	Not found
Severity	MEDIUM

5.2.4 Sensitive Information Disclosure

Description	Examine whether sensitive information disclosure issue present in the code
Result	Not found
Severity	MEDIUM

5.2.5 System API

Description	Examine whether system API application issue present in the code
Result	Not found
Severity	LOW

6. DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without ExVul's prior written consent.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts ExVul to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. ExVul's position is that each company and individual are responsible for their own due diligence and continuous security. ExVul's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

7. REFERENCES

[1] MITRE. CWE-191: Integer Underflow (Wrap or Wraparound). https://cwe.mitre.org/data/definitions/191.html.

[2] MITRE. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197.html.

[3] MITRE. CWE-400: Uncontrolled Resource Consumption. https://cwe.mitre.org/data/definitions/400.html.

[4] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440.html.

[5] MITRE. CWE-684: Protection Mechanism Failure. https://cwe.mitre.org/data/definitions/693.html.

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

[7] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE CATEGORY: Resource Management Errors. https://cwe.mitre.org/data/definitions/399.html.

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

8. About Exvul Security

Premier Security for the Web3 Ecosystem

ExVul is a premier Web3 security firm committed to forging a secure and trustworthy decentralized ecosystem. Our elite team consists of security veterans from world-leading technology and blockchain security firms, including Huawei, YBB Captical, Qihoo 360, Amber, ByteDance, MoveBit, and PeckShield. Team member Nolan is ranked as a top-40 whitehat on Immunefi and is the platform's sole All-Star in the APAC region.

Our expertise covers the full spectrum of Web3 security. We conduct **meticulous smart contract audits**, having fortified thousands of projects on chains like Evm, Solana, Aptos, Sui etc. Our **Blockchain Protocol Audits** secure the core infrastructure of L1/L2 by uncovering deep-seated vulnerabilities. We also offer **comprehensive wallet audits** to protect user assets and provide **proactive web3 pentest**, enabling partners to neutralize threats before they strike.

Trusted by industry leaders, ExVul is the security partner for **OKX, Bitget, Cobo, Infini, Stacks, Aptos, Sui, CoreDAO, Sei** etc.

Contact

Website
www.exvul.com

Email
contact@exvul.com

Twitter
@EXVULSEC

Github
github.com/EXVUL-Sec