CredShields

Smart Contract Audit

Oct 31st, 2022 • CONFIDENTIAL

Description

This document details the process and result of the Juno Coin smart contract audit performed by CredShields Technologies PTE. LTD. on behalf of Juno between Oct 4th, 2022, and Oct 13th, 2022. And a retest was performed on 30th Oct 2022.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Juno Finance

Table of Contents

CRED SHIELDS

1. Executive Summary

Juno engaged CredShields to perform a smart contract audit from Oct 4th, 2022, to Oct 13th, 2022. During this timeframe, twelve (12) vulnerabilities were identified. **A retest was performed on 30th Oct 2022, and all the bugs have been addressed.**

During the audit, zero (0) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Juno" and should be prioritized for remediation, and fortunately, none were found.

The table below shows the in-scope assets and a breakdown of findings by severity per asset. Section 2.3 contains more information on how severity is calculated.

Assets in Scope	Critical	High	Medium	Low	info	Gas	Σ
Juno	0	0	2	2	2	6	**12**
	0	**0**	**2**	**2**	**2**	**6**	**12**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in Juno's scope during the testing window while abiding by the policies set forth by Juno's team.

State of Security

Maintaining a healthy security posture requires constant review and refinement of existing security processes. Running a CredShields continuous audit allows Juno's internal security team and development team to not only uncover specific vulnerabilities but gain a better understanding of the current security threat landscape.

We recommend running regular security assessments to identify any vulnerabilities introduced after Juno introduces new features or refactors the code.

Reviewing the remaining resolved reports for a root cause analysis can further educate Juno's internal development and security teams and allow manual or automated procedures to be put in place to eliminate entire classes of vulnerabilities in the future. This proactive approach helps contribute to future-proofing the security posture of Juno assets.

2. Methodology

Juno engaged CredShields to perform a smart contract audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation phase

CredShields team read all the provided documents and comments in the smart-contract code to understand the contract's features and functionalities. The team reviewed all the functions and prepared a mind map to review for possible security vulnerabilities in the order of the function with more critical and business-sensitive functionalities for the refactored code.

The team deployed a self-hosted version of the smart contract to verify the assumptions and validate the vulnerabilities during the audit phase.

A testing window from Oct 4th, 2022, to Oct 13th, 2022, was agreed upon during the preparation phase.

2.1.1 Scope

During the preparation phase, the following scope for the engagement was agreed-upon:

IN SCOPE ASSETS
https://github.com/bankonjuno/juno-coin

Table: List of Files in Scope

2.1.2 Documentation

Documentation was not required as the code was self-sufficient for understanding the project.

2.1.3 Audit Goals

CredShields' methodology uses individual tools and methods; however, tools are just used for aids. The majority of the audit methods involve manually reviewing the smart contract source code. The team followed the standards of the SWC registry for testing along with an extended self-developed checklist based on industry standards, but it was not limited to it. The team focused heavily on understanding the core concept behind all the functionalities along with preparing test and edge cases. Understanding the business logic and how it could have been exploited.

The audit's focus was to verify that the smart contract system is secure, resilient, and working according to its specifications. Breaking the audit activities into the following three categories:

- **Security** - Identifying security-related issues within each contract and the system of contracts.
- **Sound Architecture** - Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
- **Code Correctness and Quality** - A full review of the contract source code. The primary areas of focus include:
    - Correctness
    - Readability
    - Sections of code with high complexity
    - Improving scalability
    - Quantity and quality of test coverage

2.2 Retesting phase

Juno is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

2.3 Vulnerability classification and severity

Discovering vulnerabilities is important, but estimating the associated risk to the business is just as important.

To adhere to industry guidelines, CredShields follows OWASP's Risk Rating Methodology. This is calculated using two factors - **Likelihood** and **Impact**. Each of these parameters can take three values - **Low**, **Medium**, and **High**.

These depend upon multiple factors such as Threat agents, Vulnerability factors (Ease of discovery and exploitation, etc.), and Technical and Business Impacts. The likelihood and the impact estimate are put together to calculate the overall severity of the risk.

CredShields also define an **Informational** severity level for vulnerabilities that do not align with any of the severity categories and usually have the lowest risk involved.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

Overall, the categories can be defined as described below -

1. **Informational**

We believe in the importance of technical excellence and pay a great deal of attention to its details. Our coding guidelines, practices, and standards help ensure that our software is stable and reliable.

Informational vulnerabilities should not be a cause for alarm but rather a chance to improve the quality of the codebase by emphasizing readability and good practices.

They do not represent a direct risk to the Contract but rather suggest improvements and the best practices that can not be categorized under any of the other severity categories.

Code maintainers should use their own judgment as to whether to address such issues.

2. Low

Vulnerabilities in this category represent a low risk to the Smart Contract and the organization. The risk is either relatively small and could not be exploited on a recurring basis, or a risk that the client indicates is not important or significant, given the client's business circumstances.

3. Medium

Medium severity issues are those that are usually introduced due to weak or erroneous logic in the code.

These issues may lead to exfiltration or modification of some of the private information belonging to the end-user, and exploitation would be detrimental to the client’s reputation under certain unexpected circumstances or conditions. These conditions are outside the control of the adversary.

These issues should eventually be fixed under a certain timeframe and remediation cycle.

4. High

High severity vulnerabilities represent a greater risk to the Smart Contract and the organization. These vulnerabilities may lead to a limited loss of funds for some of the end-users.

They may or may not require external conditions to be met, or these conditions may be manipulated by the attacker, but the complexity of exploitation will be higher.

These vulnerabilities, when exploited, will impact the client's reputation negatively.

They should be fixed immediately.

5. Critical

Critical issues are directly exploitable bugs or security vulnerabilities. These issues do not require any external conditions to be met.

The majority of vulnerabilities of this type involve a loss of funds and Ether from the Smart Contracts and/or from their end-users.

The issue puts the vast majority of, or large numbers of, users' sensitive information at risk of modification or compromise.

The client's reputation will suffer a severe blow, or there will be serious financial repercussions.

Considering the risk and volatility of smart contracts and how they use gas as a method of payment to deploy the contracts and interact with them, gas optimization becomes a major point of concern. To address this, CredShields also introduces another severity category called "**Gas Optimization**" or "**Gas**". This category deals with code optimization techniques and refactoring, due to which Gas can be conserved.

2.4 CredShields staff

The following individual at CredShields managed this engagement and produced this report:

- **Shashank, Co-founder CredShields**
    - shashank@CredShields.com

Please feel free to contact this individual with any questions or concerns you have around the engagement or this document.

3. Findings

This chapter contains the results of the security assessment. Findings are sorted by their severity and grouped by the asset and SWC classification. Each asset section will include a summary. The table in the executive summary contains the total number of identified security vulnerabilities per asset per risk indication.

3.1 Findings Overview

3.1.1 Vulnerability Summary

During the security assessment, Twelve (12) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Functions should be declared External	Gas	Gas Optimization
Large Number Literals	Gas	Gas & Missing Best Practices
Outdated Pragma	Low	Outdated Compiler Version (SWC-102)
Require with Empty Message	Informational	Code optimization
Use of SafeMath	Gas	Gas Optimization
Dead/Unreachable Code	Informational	Code With No Effects - SWC-135

Use Require instead of If & Revert	Gas	Gas Optimization
Gas Optimization in Increments	Gas	Gas Optimization
Missing zero Address Validations	Low	Input Validation
Approve Fruntrunning Attack	Medium	Frontrunning
Vulnerable Components and Libraries	Medium	Components with known vulnerabilities
Gas Optimization in Require Statements	Gas	Gas Optimization

Table: Findings in Smart Contracts

3.1.2 Findings Summary

SWC ID	SWC Checklist	Test Result	Notes
SWC-100	Function Default Visibility	Not Vulnerable	Not applicable after v0.5.X (Currently using solidity v >= 0.8.6)
SWC-101	Integer Overflow and Underflow	Not Vulnerable	No such case was found
SWC-102	Outdated Compiler Version	Vulnerable	Older version is used
SWC-103	Floating Pragma	Not Vulnerable	Strict pragma is used
SWC-104	Unchecked Call Return Value	Not Vulnerable	call() is not used
SWC-105	Unprotected Ether Withdrawal	Not Vulnerable	Appropriate function modifiers and require validations are used on sensitive functions that allow token or ether withdrawal.
SWC-106	Unprotected SELFDESTRUCT Instruction	Not Vulnerable	selfdestruct() is not used anywhere
SWC-107	Reentrancy	Not Vulnerable	No notable functions were vulnerable to it.
SWC-108	State Variable Default Visibility	Not Vulnerable	Not Vulnerable
SWC-109	Uninitialized Storage Pointer	Not Vulnerable	Not vulnerable after compiler version, v0.5.0
SWC-110	Assert Violation	Not Vulnerable	Asserts are not in use.

SWC-111	Use of Deprecated Solidity Functions	Not Vulnerable	None of the deprecated functions like block.blockhash(), msg.gas, throw, sha3(), callcode(), suicide() are in use
SWC-112	Delegatecall to Untrusted Callee	Not Vulnerable	Not Vulnerable.
SWC-113	DoS with Failed Call	Not Vulnerable	No such function was found.
SWC-114	Transaction Order Dependence	Not Vulnerable	Not Vulnerable.
SWC-115	Authorization through tx.origin	Not Vulnerable	tx.origin is not used anywhere in the code
SWC-116	Block values as a proxy for time	Not Vulnerable	Block.timestamp is not used
SWC-117	Signature Malleability	Not Vulnerable	Not used anywhere
SWC-118	Incorrect Constructor Name	Not Vulnerable	All the constructors are created using the constructor keyword rather than functions.
SWC-119	Shadowing State Variables	Not Vulnerable	Not applicable as this won't work during compile time after version 0.6.0
SWC-120	Weak Sources of Randomness from Chain Attributes	Not Vulnerable	Random generators are not used.
SWC-121	Missing Protection against Signature Replay Attacks	Not Vulnerable	No such scenario was found
SWC-122	Lack of Proper Signature Verification	Not Vulnerable	Not used anywhere

SWC-123	Requirement Violation	Not Vulnerable	Not vulnerable
SWC-124	Write to Arbitrary Storage Location	Not Vulnerable	No such scenario was found
SWC-125	Incorrect Inheritance Order	Not Vulnerable	No such scenario was found
SWC-126	Insufficient Gas Griefing	Not Vulnerable	No such scenario was found
SWC-127	Arbitrary Jump with Function Type Variable	Not Vulnerable	Jump is not used.
SWC-128	DoS With Block Gas Limit	Not Vulnerable	Not Vulnerable.
SWC-129	Typographical Error	Not Vulnerable	No such scenario was found
SWC-130	Right-To-Left-Override control character (U+202E)	Not Vulnerable	No such scenario was found
SWC-131	Presence of unused variables	Not Vulnerable	No such scenario was found
SWC-132	Unexpected Ether balance	Not Vulnerable	No such scenario was found
SWC-133	Hash Collisions With Multiple Variable Length Arguments	Not Vulnerable	abi.encodePacked() or other functions are not used.
SWC-134	Message call with hardcoded gas amount	Not Vulnerable	Not used anywhere in the code
SWC-135	Code With No Effects	Vulnerable	Dead code was found
SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

4. Remediation Status

Juno is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on 30th Oct 2022, and all the issues have been addressed.** Also, the table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Functions should be declared External	Gas	**Won't Fix**
Large Number Literals	Gas	**Won't Fix**
Outdated Pragma	Low	**Won't Fix**
Require with Empty Message	Informational	**Won't Fix**
Use of SafeMath	Gas	**Won't Fix**
Dead/Unreachable Code	Informational	**Won't Fix**
Use Require instead of If & Revert	Gas	**Won't Fix**
Gas Optimization in Increments	Gas	**Won't Fix**
Missing zero Address Validations	Low	**Won't Fix**
Approve Fruntrunning Attack	Medium	**Won't Fix**
Vulnerable Components and Libraries	Medium	**Won't Fix**
Gas Optimization in Require Statements	Gas	**Won't Fix**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [Won't Fix]

Functions should be declared External

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Public functions that are never called by a contract should be declared external in order to conserve gas.
The following functions were declared as public but were not called anywhere in the contract, making public visibility useless.

Affected Code
The following functions were affected -

- **configureController()** - https://github.com/bankonjuno/juno-coin/blob/master/contracts/minting/Controller.sol#L74-L85
- **removeController()** - https://github.com/bankonjuno/juno-coin/blob/master/contracts/minting/Controller.sol#L91-L102
- **setMinterManager()** - https://github.com/bankonjuno/juno-coin/blob/master/contracts/minting/MintController.sol#L99-L102

- **incrementMinterAllowance()** - https://github.com/bankonjuno/juno-coin/blob/master/contracts/minting/MintController.sol#L135-L161
- **decrementMinterAllowance()** - https://github.com/bankonjuno/juno-coin/blob/master/contracts/minting/MintController.sol#L169-L200

Impacts

Smart Contracts are required to have effective Gas usage as they cost real money and each function should be monitored for the amount of gas it costs to make it gas efficient.
"**public**" functions cost more Gas than "**external**" functions.

Remediation

Use the "**external**" state visibility for functions that are never called from inside the contract.

Retest:

Gas severity issues are going to be ignored as contract interaction from Juno's side isn't going to be very often, and gas optimizations are just best practices and not a security exploit, and the CredShields team agrees to the opinion.

Bug ID#2 [Won't Fix]

Large Number Literals

Vulnerability Type
Gas & Missing Best Practices

Severity
Gas

Description
Solidity supports multiple rational and integer literals, including decimal fractions and scientific notations. The use of very large numbers with too many digits was detected in the code that could have been optimized using a different notation, also supported by Solidity.

Affected Code

- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v3/JunoCoinV2.sol#L47

```
function decreaseTotalSupply() external whenNotPaused onlyOwner {
    totalSupply_ = totalSupply_.sub(100000);
}
```

Impacts
Having a large number of literals in the code increases the gas usage of the contract while its deployment and when the functions are used or called from the contract.
It also makes the code harder to read and audit and increases the chances of introducing code errors.

Remediation
Scientific notation in the form of 2e10 is also supported, where the mantissa can be fractional, but the exponent has to be an integer. The literal MeE is equivalent to M *

10**E. Examples include 2e10, 2e10, 2e-10, 2.5e1, as suggested in official solidity documentation.
https://docs.soliditylang.org/en/latest/types.html#rational-and-integer-literals
The numbers can also be represented by using underscores between them to make them more readable such as “35_00_00_000”.
In this case, the number **100000** can be represented as **1E5**.

Retest:
Gas severity issues are going to be ignored as contract interaction from Juno’s side isn’t going to be very often, and gas optimizations are just best practices and not a security exploit, and the CredShields team agrees to the opinion.

Bug ID#3 [Won't Fix]

Outdated Pragma

Vulnerability Type
Outdated Compiler Version (SWC-102)

Severity
Low

Description
Using an outdated compiler version can be problematic, especially if there are publicly disclosed bugs and issues that affect the current compiler version.
The contracts found in the repository were allowing a really old compiler version to be used, i.e., **^0.6.12**.

Affected Code

- All the contracts

Impacts
If the smart contract gets compiled and deployed with an older version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions.
A complete list of vulnerabilities found in the compiler versions can be seen here - https://github.com/ethereum/solidity/blob/develop/docs/bugs.json

Remediation
It is recommended to use a recent version of the Solidity compiler that should not be the most recent version; and it should not be an outdated version as well. Using very old versions of Solidity prevents the benefits of bug fixes and newer security checks. Consider using solidity version **0.8.7**, which patches most Solidity vulnerabilities.

Retest:

The team is using forked USDC's contract and deployed their own wrapper contract on top of it. The Credshields team also checked vulnerabilities in the older compiler version and it doesn't impact the contract.

Bug ID#4 [Won't Fix]

Require with Empty Message

Vulnerability Type
Code optimization

Severity
Informational

Description
During analysis; multiple **require** statements were detected with empty messages. The statement takes two parameters, and the message part is optional. This is shown to the user when and if the **require** statement evaluates to false. This message gives more information about the conditional and why it gave a false response.

Affected Code

- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v2/FiatTokenV2.sol#L47
- https://github.com/bankonjuno/juno-coin/blob/master/contracts/upgradeability/AdminUpgradeabilityProxy.sol#L137
- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v2/FiatTokenV2_1.sol#L42

```
require(initialized && _initializedVersion == 0); //FiatTokenV2.sol
require(success); //AdminUpgradeabilityProxy.sol
require(_initializedVersion == 1); //FiatTokenV2_1.sol
```

Impacts
Having a short descriptive message in the **require** statement gives users and developers more details as to why the conditional statement failed and helps in debugging the transactions.

Remediation

It is recommended to add a descriptive message, no longer than 32 bytes, inside the **require** statement to give more detail to the user about why the condition failed.

Retest

This is more of a best practice than a security vulnerability and hence it won't be fixed.

Bug ID#5 [Won't Fix]

Use of SafeMath

Vulnerability Type
Gas Optimization

Severity
Gas

Description
SafeMath library is found to be used in the contract. This increases gas consumption more than traditional methods and validations if done manually.

Also, Solidity **0.8.0** and above includes checked arithmetic operations by default, and this renders SafeMath unnecessary.

Affected Code:

- https://github.com/bankonjuno/juno-coin/blob/master/contracts/minting/MintController.sol#L41
- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v1/FiatTokenV1.sol#L38
- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v2/upgrader/V2_1Upgrader.sol#L39
- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v2/upgrader/V2Upgrader.sol#L44
- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v3/JunoCoinV2.sol#L38
- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v3/V3Upgrader.sol

```
import { SafeMath } from "@openzeppelin/contracts/math/SafeMath.sol";
```

Impacts
This increases the gas usage of the contract.

Remediation:
We do not recommend using SafeMath library for all arithmetic operations. It is good practice to use explicit checks where it is really needed and to avoid extra checks where overflow/underflow is impossible.

The compiler should be upgraded to Solidity version **0.8.0+,** which automatically checks for overflows and underflows.

Retest
Gas severity issues are going to be ignored as contract interaction from Juno's side isn't going to be very often, and gas optimizations are just best practices and not a security exploit, and the CredShields team agrees to the opinion.

Bug ID#6 [Won't Fix]

Dead/Unreachable Code

Vulnerability Type
Code With No Effects - SWC-135

Severity
Informational

Description
It is recommended to keep the production repository clean to prevent confusion and the introduction of vulnerabilities. The functions and parameters that are never used or called externally or from inside the contracts should be removed when the contract is deployed on the mainnet.

Vulnerable Code

- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v3/JunoCoinV2.sol#L50-L52

```
    function returnSarvad() external view returns (string memory)
{
        return "sarvad";
    }
```

Impacts
This does not impact the security aspect of the Smart contract but prevents confusion when the code is sent to other developers or auditors to understand and implement. This reduces the overall size of the contracts and also helps in saving gas.

Remediation
Delete the external function "`returnSarvad()`" if it is not supposed to be used or has no real-world use case.

Retest

Juno team informed the CredShields team that the contract in which this function resides is never deployed, so they are ignoring this issue. This claim was verified by the CredShields team on the deployed contract and they found the claim to be true.

Bug ID#7 [Won't Fix]

Use Require instead of If & Revert

Vulnerability Type
Gas optimization

Severity
Gas optimization

Description
The contract **FiatTokenUtil.sol** is using a combination of if and revert statements on Line 166. This is unnecessary and increases gas usage. This can be optimized by using a require statement instead of revert.

Vulnerable Code

- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v2/FiatTokenUtil.sol#L166-L168

```
        if (returnData.length < 100) {
            revert("FiatTokenUtil: call failed");
        }
```

Impacts
Using both if and revert simultaneously costs more than than using a simple require statement. If require was used, the contract would have saved approximately 216 gas units.

Remediation
It is recommended to switch to a require statement as shown below:

```
require(returnData.length < 100, "FiatTokenUtil: call failed");
```

Retest

Gas severity issues are going to be ignored as contract interaction from Juno's side isn't going to be very often, and gas optimizations are just best practices and not a security exploit, and the CredShields team agrees to the opinion.

Bug ID#8 [Won't Fix]

Gas Optimization in Increments

Vulnerability Type
Gas optimization

Severity
Gas optimization

Description
The contract **FiatTokenUtil.sol** is using a for loop on Line which is using post increments for the variable "**i**".
The contract can save some gas by changing this to **++i**.
++i costs less gas compared to **i++** or **i += 1** for unsigned integers. In **i++**, the compiler has to create a temporary variable to store the initial value. This is not the case with **++i** in which the value is directly incremented and returned, thus, making it a cheaper alternative.

Vulnerable Code

- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v2/FiatTokenUtil.sol#L80

```
for (uint256 i = 0; i < num; i++) {...}
```

Impacts
Using i++ instead of ++i costs the contract deployment around 432 more gas units.

Remediation
It is recommended to switch to **++i** and change the code accordingly so the function logic remains the same and meanwhile save some gas.

Retest

Gas severity issues are going to be ignored as contract interaction from Juno's side isn't going to be very often, and gas optimizations are just best practices and not a security exploit, and the CredShields team agrees to the opinion.

Bug ID#9 [Won't Fix]

Missing zero Address Validations

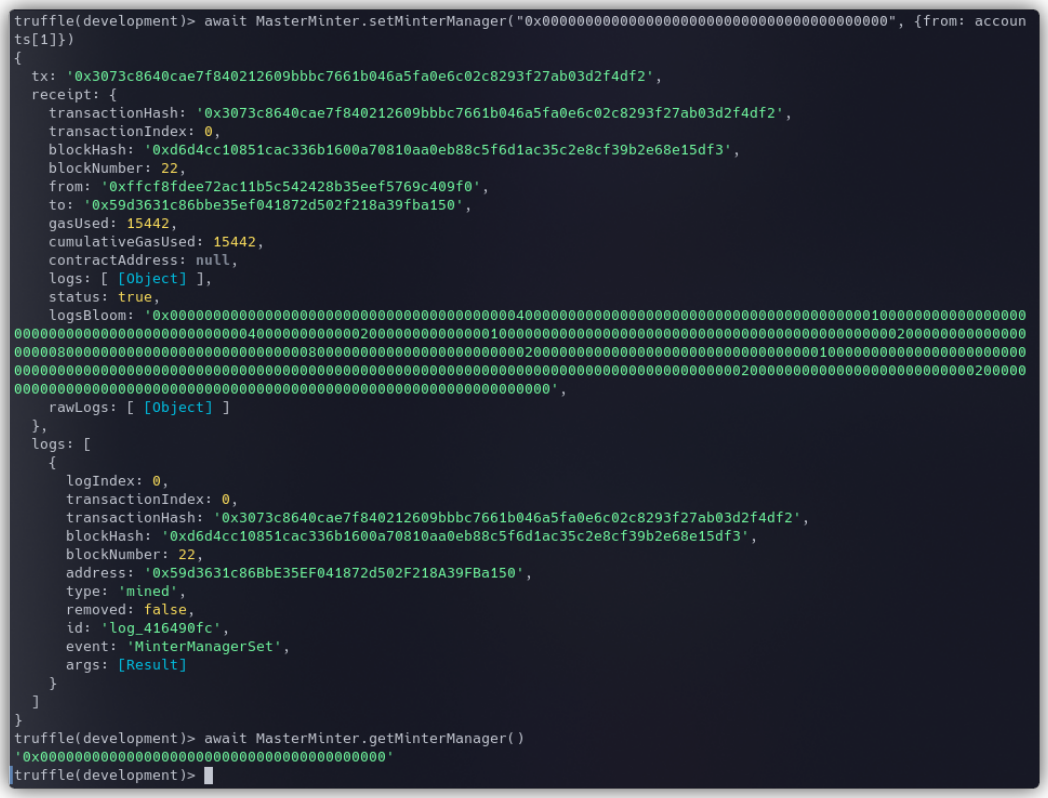
Vulnerability Type
Input validation

Severity
Low

Description
Every critical address change, sensitive functions handling addresses and tokens, and all the other address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burned forever if an invalid or a zero address is supplied by mistake or due to contract errors.

Vulnerable Code

- getMinterManager() - https://github.com/bankonjuno/juno-coin/blob/master/contracts/minting/MintController.sol#L99-L102



- rescueERC20() - https://github.com/bankonjuno/juno-coin/blob/master/contracts/v1.1/Rescuable.sol#L60-L66
- address newProxyAdmin, address newOwner inside constructor - https://github.com/bankonjuno/juno-coin/blob/master/contracts/v2/upgrader/V2Upgrader.sol#L60-L74
- address fiatTokenProxy inside constructor - https://github.com/bankonjuno/juno-coin/blob/master/contracts/v2/upgrader/V2UpgraderHelper.sol#L43-L45
- address newProxyAdmin, address newOwner inside constructor - https://github.com/bankonjuno/juno-coin/blob/master/contracts/v2/upgrader/V2_1Upgrader.sol#L55-L69
- address fiatToken in constructor - https://github.com/bankonjuno/juno-coin/blob/master/contracts/v2/FiatTokenUtil.sol#L45-L47

- address newProxyAdmin, address newOwner inside constructor - https://github.com/bankonjuno/juno-coin/blob/master/contracts/v3/V3Upgrader.sol#L44-L56

Impacts

Missing a zero-address validation on sensitive address changes and token transfers may cause contract ownership to be los, and the tokens may be burned forever.

Remediation

Add a zero-address validation to the functions specified above.

Retest

There is no major possible exploitation of this bug and hence it will be left as it is.

Bug ID#10 [Won't Fix]

Approve Fruntrunning Attack

Vulnerability Type
Frontrunning

Severity
Medium

Description
The contract **FiatTokenV1.sol** is defining an "_approve" function on line 226 which is vulnerable to front-running attack.
Approve is well known to be vulnerable to front-running attacks. This may be exploited in cases where in case the user decides to modify the spending amount in quick succession and the attacker sees the change and transfers more tokens than required.
The exploitation involves two parties and attacker must be monitoring the transactions.

Vulnerable Code

- https://github.com/bankonjuno/juno-coin/blob/master/contracts/v1/FiatTokenV1.sol#L208-L218

```
    function _approve(
        address owner,
        address spender,
        uint256 value
    ) internal override {
        require(owner != address(0), "ERC20: approve from the zero
address");
        require(spender != address(0), "ERC20: approve to the zero
address");
        allowed[owner][spender] = value;
        emit Approval(owner, spender, value);
    }
```

Impacts

Front-running attacks might allow attackers to front-run a user transaction and withdraw/transfer more tokens than the victim initially intended to allow.

Remediation

Instead of _approve() to change the allowance, it is recommended to use increaseAllowance and decreaseAllowance functions which are meant for this use case. It is also recommended to refer to the following documentation for more information: https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit

Retest

This function resides in an older implementation contract which has been upgraded to fix this issue, and the CredShields team verified the claim as in the upgraded version of the contract increase increaseAllowance and decreaseAllowance is used.

Bug ID#11[Won't Fix]

Vulnerable Components and Libraries

Vulnerability Type
Components with known vulnerabilities

Severity
Medium

Description
The Juno Coin project uses various libraries and modules which were found to be outdated and most of them were affected by publicly disclosed exploits and vulnerabilities.

PoC

1. Run the following command in the project directory to list out the results - "yarn audit". A screenshot is shown below:

```
~/Desktop/CredShields/Audits/Juno/juno-coin ) on  master ↓3 !8 ?1
yarn audit
yarn audit v1.22.19
```

high	Improper Initialization in OpenZeppelin
Package	@openzeppelin/contracts
Patched in	>=4.4.1
Dependency of	@openzeppelin/contracts
Path	@openzeppelin/contracts
More info	https://www.npmjs.com/advisories/1067409

moderate	OpenZeppelin Contracts ERC165Checker unbounded gas consumption
Package	@openzeppelin/contracts
Patched in	>=4.7.2
Dependency of	@openzeppelin/contracts
Path	@openzeppelin/contracts
More info	https://www.npmjs.com/advisories/1083165

high	Insecure serialization leading to RCE in serialize-javascript
Package	serialize-javascript
Patched in	>=3.1.0
Dependency of	truffle
Path	truffle > mocha > serialize-javascript
More info	https://www.npmjs.com/advisories/1069320

high	Inefficient Regular Expression Complexity in chalk/ansi-regex
Package	ansi-regex
Patched in	>=3.0.1
Dependency of	truffle
Path	truffle > mocha > wide-align > string-width > strip-ansi > ansi-regex
More info	https://www.npmjs.com/advisories/1081982

Impacts

A total of 179 vulnerabilities were found in 1162 packages that were audited. The severity ratings are 12 Low, 53 Moderate, 88 High, and 26 Critical.

Remediation

As a best practice, keep all software up to date, especially if there exists a known vulnerability or weakness associated with an older version.

Retest

This is just used for deployment and hence does not have any impact. The CredShields team

Bug ID#12 [Won't Fix]

Gas Optimization in Require Statements

Vulnerability Type
Frontrunning

Severity
Gas

Description
The **require()** statement takes an input string to show errors if the validation fails.
This strings inside these functions that are longer than **32 bytes** require at least one additional MSTORE, along with additional overhead for computing memory offset, and other parameters. For this purpose, having the strings lesser than 32 bytes saves significant amount of gas.

Vulnerable Code

- Affects all require statements with more than 32 characters.

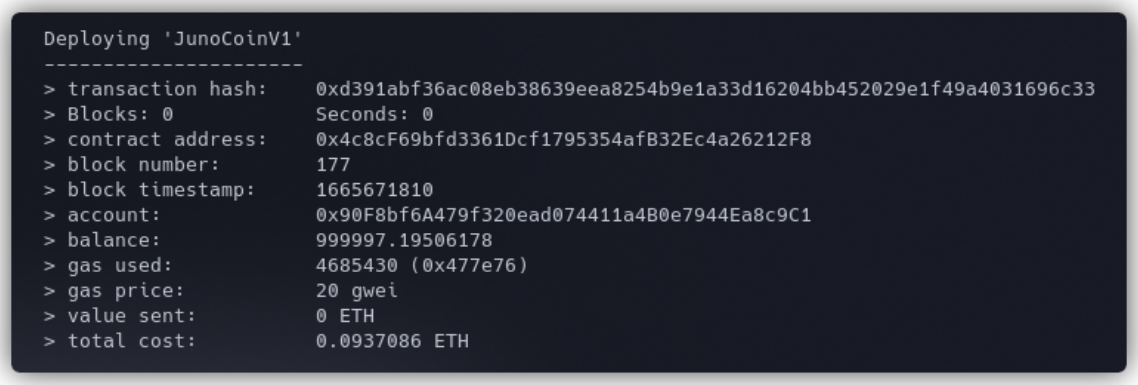
PoC

1. The following screenshot is taken after a normal deployment without changing anything:

2. The following screenshot is taken after shortening the string inside the require statement to 32 bytes:

```
Deploying 'JunoCoinV1'
----------------------
> transaction hash:    0x88ede0a30fb82d3e34150d3a79912df236995c4dce2729b03b99e71021f0d864
> Blocks: 0            Seconds: 0
> contract address:    0xdC78afe9cFDe0576Ff236667DC8c380615c24Ca9
> block number:        82
> block timestamp:     1665671164
> account:             0x90F8bf6A479f320ead074411a4B0e7944Ea8c9C1
> balance:             999998.698444
> gas used:            4683078 (0x477546)
> gas price:           20 gwei
> value sent:          0 ETH
> total cost:          0.09366156 ETH
```

3. The following screenshot is taken after shortening the string inside the require statement to 33 bytes:



```
Deploying 'JunoCoinV1'
----------------------
> transaction hash:    0xd391abf36ac08eb38639eea8254b9e1a33d16204bb452029e1f49a4031696c33
> Blocks: 0            Seconds: 0
> contract address:    0x4c8cF69bfd3361Dcf1795354afB32Ec4a26212F8
> block number:        177
> block timestamp:     1665671810
> account:             0x90F8bf6A479f320ead074411a4B0e7944Ea8c9C1
> balance:             999997.19506178
> gas used:            4685430 (0x477e76)
> gas price:           20 gwei
> value sent:          0 ETH
> total cost:          0.0937086 ETH
```

The significant gas difference can be seen in the screenshots above when the characters were changed from 32 to 33.

Impacts

Having longer require strings than 32 bytes cost a significant amount of gas.

Remediation

It is recommended to short the strings passed inside **require()** statements to fit under **32 bytes**. This will decrease the gas usage at the time of deployment and at runtime when the validation condition is met.

This is just one of the examples. It is recommended to go though the code and find all the occurrences that are longer than 32 bytes.

Retest

Gas severity issues are going to be ignored as contract interaction from Juno's side isn't going to be very often, and gas optimizations are just best practices and not a security exploit, and the CredShields team agrees to the opinion.

6. Disclosure

The Reports neither endorse nor condemn any specific project or team, nor do they guarantee the security of any specific project. The contents of this report do not, and should not be interpreted as having any bearing on, the economics of tokens, token sales, or any other goods, services, or assets.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. There is no warranty or representation made by this report to any Third Party in regards to the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business.
In no way should a third party use these reports to make any decisions about buying or selling a token, product, service, or any other asset. It should be noted that this report is not investment advice, is not intended to be relied on as investment advice, and has no endorsement of this project or team. It does not serve as a guarantee of the project's absolute security.
CredShields Audit team owes no duty to any third party by virtue of publishing these Reports.