



# CredShields

# Smart Contract Audit

---

**Feb 7th, 2023 • CONFIDENTIAL**

## **Description**

This document details the process and result of the Digital Invoice Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Obius between Feb 11th, 2023, and Feb 18th, 2023. **And a retest was performed on Feb 28th, 2023.**

## **Author**

Shashank (Co-founder, CredShields)

[shashank@CredShields.com](mailto:shashank@CredShields.com)

## **Reviewers**

Aditya Dixit (Research Team Lead)

[aditya@CredShields.com](mailto:aditya@CredShields.com)

## **Prepared for**

Obius

# Table of Contents

<b>1. Executive Summary</b>	<b>3</b>
State of Security	4
<b>2. Methodology</b>	<b>5</b>
2.1 Preparation phase	5
2.1.1 Scope	6
2.1.2 Documentation	6
2.1.3 Audit Goals	6
2.2 Retesting phase	7
2.3 Vulnerability classification and severity	7
2.4 CredShields staff	10
<b>3. Findings</b>	<b>11</b>
3.1 Findings Overview	11
3.1.1 Vulnerability Summary	11
3.1.2 Findings Summary	13
<b>4. Remediation Status</b>	<b>17</b>
<b>5. Bug Reports</b>	<b>19</b>
Bug ID#1 [Fixed]	19
Floating and Outdated Pragma versions	19
Bug ID#2 [Fixed]	21
Unused Variables	21
Bug ID#3 [Fixed]	23
Functions should be declared External	23
Bug ID#4 [Fixed]	25
Gas Optimization in Require Statements	25
Bug ID#5 [Fixed]	27
Custom Errors instead of Revert	27
Bug ID#6 [Fixed]	29
Missing Price Feed Validation	29
Bug ID#7 [Fixed]	31

Gas Optimization with Counters	31
Bug ID#8 [Won't Fix]	33
Missing Input Validation in Prefix-URL	33
Bug ID#9 [Fixed]	35
Missing Input Validation in Metadata	35
Bug ID#10 [Fixed]	37
Improper Function Visibility and Missing Validation	37
Bug ID#11 [Fixed]	39
Missing Input Validation in Fee Percentage	39
Bug ID#12 [Fixed]	41
Missing Events in important functions	41
Bug ID#13 [Fixed]	43
Private Invoice's Information Disclosure	43
Bug ID#14 [Fixed]	45
Incorrect Invoice Price Calculation	45
6. Disclosure	47

# 1. Executive Summary

Obius engaged CredShields to perform a smart contract audit from Feb 11th, 2023, to Feb 18th, 2023. During this timeframe, Ten (10) vulnerabilities were identified. **A retest was performed on Feb 28th, 2023, and all the bugs have been addressed.**

During the audit, one (1) vulnerability was found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Obius" and should be prioritized for remediation.

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
Digital Invoice Smart Contract	0	1	1	3	1	4	<b>10</b>
	<b>0</b>	<b>1</b>	<b>1</b>	<b>3</b>	<b>1</b>	<b>4</b>	<b>10</b>

*Table: Vulnerabilities Per Asset in Scope*

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

## State of Security

To maintain a robust security posture, it is essential to continuously review and improve upon current security processes. Utilizing CredShields' continuous audit feature allows both Obius's internal security and development teams to not only identify specific vulnerabilities, but also gain a deeper understanding of the current security threat landscape.

To ensure that vulnerabilities are not introduced when new features are added, or code is refactored, we recommend conducting regular security assessments. Additionally, by analyzing the root cause of resolved vulnerabilities, the internal teams at Obius can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Obius can future-proof its security posture and protect its assets.

## 2. Methodology

---

Obius engaged CredShields to perform an Obius Smart Contract audit. The following sections cover how the engagement was put together and executed.

### 2.1 Preparation phase

The CredShields team meticulously reviewed all provided documents and comments in the smart-contract code to gain a thorough understanding of the contract's features and functionalities. They meticulously examined all functions and created a mind map to systematically identify potential security vulnerabilities, prioritizing those that were more critical and business-sensitive for the refactored code. To confirm their findings, the team deployed a self-hosted version of the smart contract and performed verifications and validations during the audit phase.

A testing window from Feb 11th, 2023, to Feb 18th, 2023, 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
<a href="https://github.com/Akestor/Digital-Invoice">https://github.com/Akestor/Digital-Invoice</a>

*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 uses both in-house tools and manual methods for comprehensive smart contract security auditing. The majority of the audit is done by manually reviewing the contract source code, following SWC registry standards, and an extended industry standard self-developed checklist. The team places emphasis on understanding core concepts, preparing test cases, and evaluating business logic for potential vulnerabilities.

## 2.2 Retesting phase

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

## 2.3 Vulnerability classification and severity

CredShields follows OWASP's Risk Rating Methodology to determine the risk associated with discovered vulnerabilities. This approach considers two factors - Likelihood and Impact - which are evaluated with three possible values - **Low**, **Medium**, and **High**, based on factors such as Threat agents, Vulnerability factors, Technical and Business Impacts. The overall severity of the risk is calculated by combining the likelihood and impact estimates.

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 prioritize technical excellence and pay attention to detail in our coding practices. Our guidelines, standards, and best practices help ensure software stability and reliability. Informational vulnerabilities are opportunities for improvement and do



not pose a direct risk to the contract. Code maintainers should use their own judgment on whether to address them.

## **2. Low**

Low-risk vulnerabilities are those that either have a small impact or can't be exploited repeatedly or those the client considers insignificant based on their specific business circumstances.

## **3. Medium**

Medium-severity vulnerabilities are those caused by weak or flawed logic in the code and can lead to exfiltration or modification of private user information. These vulnerabilities can harm the client's reputation under certain conditions and should be fixed within a specified timeframe.

## **4. High**

High-severity vulnerabilities pose a significant risk to the Smart Contract and the organization. They can result in the loss of funds for some users, may or may not require specific conditions, and are more complex to exploit. These vulnerabilities can harm the client's reputation and should be fixed immediately.

## **5. Critical**

Critical issues are directly exploitable bugs or security vulnerabilities that do not require specific conditions. They often result in the loss of funds and Ether from Smart Contracts or users and put sensitive user information at risk of compromise

or modification. The client's reputation and financial stability will be severely impacted if these issues are not addressed immediately.

## **6. Gas**

To address the risk and volatility of smart contracts and the use of gas as a method of payment, CredShields has introduced a "Gas" severity category. This category deals with optimizing code and refactoring to conserve gas.

## 2.4 CredShields staff

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

- **Shashank, Co-founder CredShields**
  - [shashank@CredShields.com](mailto: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, Ten (10) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC   Vulnerability Type
Floating and Outdated Pragma versions	Low	Floating Pragma ( <a href="#">SWC-103</a> )
Unused Variables	Informational	Dead Code
Functions should be declared External	Gas	Gas Optimization
Gas Optimization in Require Statements	Gas	Gas Optimization
Custom Errors instead of Revert	Gas	Gas Optimization
Missing Price Feed Validation	Medium	Input Validation
Gas Optimization with Counters	Gas	Gas Optimization

Missing Input Validation in Prefix-URL	Low	Input Validation
Missing Input Validation in Metadata	High	Input Validation
Improper Function Visibility and Missing Validation	Low	Business Logic
Missing Input Validation in Fee Percentage	Low	Input Validation
Missing Events in important functions	Low	Missing Best Practices
Private Invoice's Information Disclosure	Medium	Information Disclosure
Incorrect Invoice Price Calculation	High	Business Logic

*Table: Findings in Smart Contracts*

### 3.1.2 Findings Summary

SWC ID	SWC Checklist	Test Result	Notes
SWC-100	<a href="#">Function Default Visibility</a>	Not Vulnerable	Not applicable after <b>v0.5.X</b> (Currently using solidity <b>v &gt;= 0.8.6</b> )
SWC-101	<a href="#">Integer Overflow and Underflow</a>	Not Vulnerable	The issue persists in versions before <b>v0.8.X</b> .
SWC-102	<a href="#">Outdated Compiler Version</a>	Not Vulnerable	Version 0 <sup>^</sup> .8.0 and above is used
SWC-103	<a href="#">Floating Pragma</a>	Not Vulnerable	Contract uses floating pragma
SWC-104	<a href="#">Unchecked Call Return Value</a>	Not Vulnerable	<b>call()</b> is not used
SWC-105	<a href="#">Unprotected Ether Withdrawal</a>	Not Vulnerable	Appropriate function modifiers and require validations are used on sensitive functions that allow token or ether withdrawal.
SWC-106	<a href="#">Unprotected SELFDESTRUCT Instruction</a>	Not Vulnerable	<b>selfdestruct()</b> is not used anywhere
SWC-107	<a href="#">Reentrancy</a>	Not Vulnerable	No notable functions were vulnerable to it.
SWC-108	<a href="#">State Variable Default Visibility</a>	Not Vulnerable	Not Vulnerable
SWC-109	<a href="#">Uninitialized Storage Pointer</a>	Not Vulnerable	Not vulnerable after compiler version, <b>v0.5.0</b>

SWC-110	<a href="#">Assert Violation</a>	Not Vulnerable	Asserts are not in use.
SWC-111	<a href="#">Use of Deprecated Solidity Functions</a>	Not Vulnerable	None of the deprecated functions like <code>block.blockhash()</code> , <code>msg.gas</code> , <code>throw</code> , <code>sha3()</code> , <code>callcode()</code> , <code>suicide()</code> are in use
SWC-112	<a href="#">Delegatecall to Untrusted Callee</a>	Not Vulnerable	Not Vulnerable.
SWC-113	<a href="#">DoS with Failed Call</a>	Not Vulnerable	No such function was found.
SWC-114	<a href="#">Transaction Order Dependence</a>	Not Vulnerable	Not Vulnerable.
SWC-115	<a href="#">Authorization through tx.origin</a>	Not Vulnerable	<code>tx.origin</code> is not used anywhere in the code
SWC-116	<a href="#">Block values as a proxy for time</a>	Not Vulnerable	<code>Block.timestamp</code> is not used
SWC-117	<a href="#">Signature Malleability</a>	Not Vulnerable	Not used anywhere
SWC-118	<a href="#">Incorrect Constructor Name</a>	Not Vulnerable	All the constructors are created using the <code>constructor</code> keyword rather than functions.
SWC-119	<a href="#">Shadowing State Variables</a>	Not Vulnerable	Not applicable as this won't work during compile time after version <code>0.6.0</code>
SWC-120	<a href="#">Weak Sources of Randomness from Chain Attributes</a>	Not Vulnerable	Random generators are not used.
SWC-121	<a href="#">Missing Protection against Signature Replay Attacks</a>	Not Vulnerable	No such scenario was found

SWC-122	<a href="#">Lack of Proper Signature Verification</a>	Not Vulnerable	Not used anywhere
SWC-123	<a href="#">Requirement Violation</a>	Not Vulnerable	Not vulnerable
SWC-124	<a href="#">Write to Arbitrary Storage Location</a>	Not Vulnerable	No such scenario was found
SWC-125	<a href="#">Incorrect Inheritance Order</a>	Not Vulnerable	No such scenario was found
SWC-126	<a href="#">Insufficient Gas Griefing</a>	Not Vulnerable	No such scenario was found
SWC-127	<a href="#">Arbitrary Jump with Function Type Variable</a>	Not Vulnerable	<b>Jump</b> is not used.
SWC-128	<a href="#">DoS With Block Gas Limit</a>	Not Vulnerable	Not Vulnerable.
SWC-129	<a href="#">Typographical Error</a>	Not Vulnerable	No such scenario was found
SWC-130	<a href="#">Right-To-Left-Override control character (U+202E)</a>	Not Vulnerable	No such scenario was found
SWC-131	<a href="#">Presence of unused variables</a>	Not Vulnerable	No such scenario was found
SWC-132	<a href="#">Unexpected Ether balance</a>	Not Vulnerable	No such scenario was found
SWC-133	<a href="#">Hash Collisions With Multiple Variable Length Arguments</a>	Not Vulnerable	<b>abi.encodePacked()</b> or other functions are not used.
SWC-134	<a href="#">Message call with hardcoded gas amount</a>	Not Vulnerable	Not used anywhere in the code
SWC-135	<a href="#">Code With No Effects</a>	Not Vulnerable	No such scenario was found
SWC-136	<a href="#">Unencrypted Private Data On-Chain</a>	Not Vulnerable	No such scenario was found





## 4. Remediation Status

Obius is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Feb 28th, 2023, and all the issues have been addressed.**

Also, the table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Floating and Outdated Pragma versions	Low	Fixed [06/02/2023]
Unused Variables	Informational	Fixed [06/02/2023]
Functions should be declared External	Gas	Fixed [06/02/2023]
Gas Optimization in Require Statements	Gas	Fixed [06/02/2023]
Custom Errors instead of Revert	Gas	Fixed [06/02/2023]
Missing Price Feed Validation	Medium	Fixed [06/02/2023]
Gas Optimization with Counters	Gas	Fixed [06/02/2023]
Missing Input Validation in Prefix-URL	Low	Fixed

		<b>[07/03/2023]</b>
Missing Input Validation in Metadata	High	<b>Fixed</b> <b>[28/02/2023]</b>
Improper Function Visibility and Missing Validation	Low	<b>Fixed</b> <b>[06/02/2023]</b>
Missing Input Validation in Fee Percentage	Low	<b>Fixed</b> <b>[28/02/2023]</b>
Missing Events in important functions	Low	<b>Fixed</b> <b>[28/02/2023]</b>
Private Invoice's Information Disclosure	Fixed	<b>Fixed</b> <b>[28/02/2023]</b>
Incorrect Invoice Price Calculation	Fixed	<b>Fixed</b> <b>[28/02/2023]</b>

*Table: Summary of findings and status of remediation*

## 5. Bug Reports

---

Bug ID#1 [**Fixed**]

### Floating and Outdated Pragma versions

#### Vulnerability Type

Floating Pragma ([SWC-103](#))

#### Severity

Low

#### Description

Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities.

The contracts found in the repository were allowing floating or unlocked pragma to be used, i.e., **^0.8.0**. This allows the contracts to be compiled with all the solidity compiler versions above the limit specified. The following contracts were found to be affected -

#### Affected Code

- ^0.8.0 - ContractV2.sol
- ^0.8.0 - Base64.sol

#### Impacts

If the smart contract gets compiled and deployed with an older or too recent version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions or unidentified exploits in the new versions.

Incompatibility issues may also arise if the contract code does not support features in other compiler versions, therefore, breaking the logic.

The likelihood of exploitation is really low therefore this is only informational.

**Remediation**

Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.9 pragma version which is stable and not too recent.

Reference: <https://swcregistry.io/docs/SWC-103>

**Retest:**

The contract now uses a fixed pragma version which is 0.8.9

## Bug ID#2 [Fixed]

### Unused Variables

#### Vulnerability Type

Dead Code

#### Severity

Informational

#### Description

It is recommended to keep the production repository clean to prevent confusion and the introduction of vulnerabilities.

The functions and parameters, contracts, and interfaces that are never used or called externally or from inside the contracts should be removed when the contract is deployed on the mainnet.

In this case, the function **getEthPrice()** defined the parameters **roundID**, **startedAt**, **timeStamp**, and **answeredInRound** which were never used or returned.

The other unused local variables were **data** and **ownData**.

#### Affected Code

- ContractV2.sol - Line 143, 144, 180

```
function getEthPrice() public view returns (int) {  
    (  
        uint80 roundID,  
        int price,  
        uint startedAt,  
        uint timeStamp,  
        uint80 answeredInRound  
    ) = priceFeed.latestRoundData();  
    return price;  
}
```

```
        (bool success, bytes memory data) = inv.from.call{value:
inv.amount-inv.cut}("");
        (bool ownSuccess, bytes memory ownData) =
payable(owner()).call{value: msg.value-(inv.amount-inv.cut)}("");
```

### 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

If the parameters are not used, it is recommended to just accept blank values like -  
(,price,,,) = priceFeed.latestRoundData();

### Retest:

The function is now accepting blank values for unused return parameters.

<https://github.com/Akestor/Digital-Invoice/blob/8ec62b225a36e3fef650634c3e210133643f1772/Contracts/ContractV2.sol#L199-L207>

## Bug ID#3 [Fixed]

### 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 -

- createInvoice() - Line 98
- payInvoice - Line 132

#### 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

The public functions have been changed to external to save gas.

<https://github.com/Akestor/Digital-Invoice/blob/8ec62b225a36e3fef650634c3e210133643f1772/Contracts/ContractV2.sol#L114>



<https://github.com/Akestor/Digital-Invoice/blob/8ec62b225a36e3fef650634c3e210133643f1772/Contracts/ContractV2.sol#L150>

## Bug ID#4 [Fixed]

# Gas Optimization in Require Statements

### Vulnerability Type

Gas Optimization

### Severity

Gas

### Description

The **require()** statement takes an input string to show errors if the validation fails.

The 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 strings lesser than 32 bytes saves a significant amount of gas.

### Vulnerable Code

- ContractV2.sol - Line 134
- ContractV2.sol - Line 233

```
require(msg.sender==inv.to, "This invoice is not generated for  
you");  
  
require(_exists(_tokenId), "ERC721Metadata: URI query for nonexistent  
token");
```

### Impacts

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

### Remediation

It is recommended to shorten 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.

### **Retest**

Errors have now been updated to a smaller value to save gas.

## Bug ID#5 [Fixed]

### Custom Errors instead of Revert

#### Vulnerability Type

Gas Optimization

#### Severity

Gas

#### Description

The contract was found to be using **revert()** statements in multiple places. Since Solidity v0.8.4, custom errors have been introduced which are a better alternative to the revert. This allows the developers to pass custom errors with dynamic data while reverting the transaction and also makes the whole implementation a bit cheaper than using revert.

#### Vulnerable Code

- ContractV2.sol - Line 112, 167

```
...  
    }else{  
        revert("Invalid payment mode");  
    }  
...
```

#### Impacts

Using revert() instead of error() costs more gas.

#### Remediation

It is recommended to replace the instances of revert() statements with error() to save gas.

#### Retest

The team is using a different method of validating errors using require statements with less than 32 bytes of data.

## Bug ID#6 [Fixed]

### Missing Price Feed Validation

#### Vulnerability Type

Input Validation

#### Severity

Medium

#### Description

Chainlink has a library **AggregatorV3Interface** with a function called **latestRoundData()**. This function returns the price feed among other details for the latest round. The contract was found to be using **latestRoundData()** without proper input validations on the returned parameters which might result in a stale and outdated price.

#### Vulnerable Code

- ContractV2.sol - Line 187

```
function getEthPrice() public view returns (int) {  
    (  
        uint80 roundID,  
        int price,  
        uint startedAt,  
        uint timeStamp,  
        uint80 answeredInRound  
    ) = priceFeed.latestRoundData();  
    return price;  
}
```

#### Impacts

Having oracles with functions to fetch price feed without any validation might introduce erroneous or invalid price values that could result in an invalid price calculation further in the contract.

## Remediation

It is recommended to have input validations for all the parameters obtained from the Chainlink price feed. Ideally, the following validations should be implemented:

```
(uint80 roundID ,answer,, uint256 timestamp, uint80 answeredInRound) =  
AggregatorV3Interface(chainLinkAggregatorMap[underlying]).latestRoundData();  
  
require(answer > 0, "Chainlink price <= 0");  
require(answeredInRound >= roundID, "Stale price");  
require(timestamp != 0, "Round not complete");
```

## Retest

The necessary price feed validations have been added to the function.

<https://github.com/Akestor/Digital-Invoice/blob/8ec62b225a36e3fef650634c3e210133643f1772/Contracts/ContractV2.sol#L199-L207>

## Bug ID#7 [Fixed]

### Gas Optimization with Counters

#### Vulnerability Type

Gas Optimization

#### Severity

Gas

#### Description

The OpenZeppelin's Counters library makes the code more readable and at the same time streamlines the process of integer increment and decrement. But this costs more gas than a traditional way of using uint directly in the contract. This can be optimized if the contract uses uint instead of Counters library.

#### Affected Code

- ContractV2.sol - Line 33

```
contract InvoiceNFT is ERC721, Ownable {  
    using Counters for Counters.Counter;
```

#### Impacts

Using OpenZeppelin's Counters library costs more gas instead of using traditional uint variables. This happens because the EVM adds more code to jump to the Counters library to execute the respective code. This also increases the total deployment size.

#### Remediation

It is recommended to use a traditional uint variable directly in the contract instead of Counters to save some gas unless Counters is absolutely necessary.

#### Retest:



Counters have been removed in favor of custom incremental logic.

## Bug ID#8 [Fixed]

### Missing Input Validation in Prefix-URL

#### Vulnerability Type

Input Validation

#### Severity

Low

#### Description

The contract accepts a prefix URL during contract creation inside the constructor and on the function **setPrefixUrl()**. This prefix is used inside the function **buildMetadata()** to build JSON metadata. The JSON object can be broken by passing invalid characters such as a double quote inside the URL.

#### Affected Code

- ContractV2.sol - Line 81, 226

```
    constructor(string memory _prefix_url,address _USDT_ADDRESS, address
_USDC_ADDRESS, address _AGGREGATOR_ADDRESS) ERC721("Invoice", "INV") {

    prefixUrl = _prefix_url;
    USDT_ADDRESS = _USDT_ADDRESS;
    USDC_ADDRESS = _USDC_ADDRESS;
    tokenUSDT = IERC20(address(_USDT_ADDRESS));
    tokenUSDC = IERC20(address(_USDC_ADDRESS));
    priceFeed = AggregatorV3Interface(_AGGREGATOR_ADDRESS);
}

function setPrefixUrl(string memory _uri) external onlyOwner{
    prefixUrl = _uri;
```

```
}
```

### **Impacts**

The missing input validation can be exploited by privileged administrators to break the JSON metadata that is generated. This will break the frontend Dapp's code wherever it is parsed.

### **Remediation**

It is recommended to have input validation on the prefix URI parameter so that it does not accept invalid strings. It can also encode certain characters that could break the JSON. Add a regex check or encode the input value.

### **Retest:**

A validation was added to the URL value

<https://github.com/lazyCoder-max/Digital-Invoice-Solidity/blob/main/ContractV2.sol#L275>

## Bug ID#9 [Fixed]

### Missing Input Validation in Metadata

#### Vulnerability Type

Input Validation

#### Severity

High

#### Description

The contract accepts metadata parameters during invoice creation inside the function **createInvoice()**. These metadata values are used inside the function **buildMetadata()** to build JSON metadata. The JSON object can be broken by passing invalid characters such as a double quote inside the parameters.

#### Affected Code

- ContractV2.sol - Line 98

```
function createInvoice(address _to, uint _paymentMode, uint _amount,  
string memory _invoiceUrl, string[] memory _meta) public  
returns(uint){...}
```

```

    "type": "Alias", "value": "1"}, {"trait_
    t_type": "Amount", "value": "0"}]]%

```

## Bug ID#10 [Fixed]

# Improper Function Visibility and Missing Validation

### Vulnerability Type

Business Logic

### Severity

Low

### Description

The contract InvoiceNFT defines a public function called **tokenURI()** that accepts a token ID which is then validated if it exists or not. If the token exists, **buildMetadata()** is called with a valid token ID. However, there's no such validation in the **buildMetadata()** function and since its visibility is public, it can be called by any external user directly to generate a fake non-existent token's invoice.

### Affected Code

- ContractV2.sol - Line 269

```
function tokenURI(uint _tokenId) public view virtual override
returns (string memory) {
    require(_exists(_tokenId), "ERC721Metadata: URI query for
nonexistent token");
    return buildMetadata(_tokenId);
}

function buildMetadata(uint _tokenId) public view returns(string
memory) {...}
```

## Impacts

The function **buildMetadata()** can be called by any external user to bypass the input validations on Line 233 to generate fake invoices for nonexistent tokens.

## Remediation

It is recommended to change the visibility of the **buildMetadata()** function to internal so that it can't be called externally. The metadata can be generated using **tokenURI()** function that already has an input validation in place.

## Retest:

This has been fixed and the function **buildMetadata()** has been marked as internal.

<https://github.com/Akestor/Digital-Invoice/blob/8ec62b225a36e3fef650634c3e210133643f1772/Contracts/ContractV2.sol#L288>

## Bug ID#11 [Fixed]

### Missing Input Validation in Fee Percentage

#### Vulnerability Type

Input Validation

#### Severity

Low

#### Description

The contract adds a payment mode using the function “**addPaymentMode()**” which also accepts a payment fee.

There’s no validation on this fee to check if the fee is not equal to zero.

#### Affected Code

- ContractV2.sol - Line 443

```
function addPaymentMode(address creatorAddress,string memory
PaymentMethodName,address PaymentMethodAddress,bool isActive,bool
isToken,uint feePercentage,bool isPublic) public
{
    require(msg.sender==contractOwner, "Method Not Allowed!");
    IERC20 candidate;
    if(length>0)
    {
        candidate = IERC20(PaymentMethodAddress);
    }
    PaymentMethod memory paymentMode =
PaymentMethod(length,creatorAddress,PaymentMethodName,PaymentMethodAddre
ss,candidate,isActive,isToken,feePercentage,isPublic);
    PAYMENT_MODE[length] = paymentMode; // adds the new payment
method
```



```
length++;  
}
```

### Impacts

Due to missing input validation, it will be possible to set the fee to 0 which will introduce errors during calculations.

### Remediation

It is recommended to have input validation on the fee percentage similar to what is done in the **updatePaymentMode()** function.

### Retest:

A check for 0 value has been added

<https://github.com/Akestor/Digital-Invoice/blob/909a8eb9b2932244c7508fb2b5891f265d656c84/Contracts/ContractV2.sol#L480>

## Bug ID#12 [Fixed]

### Missing Events in important functions

#### Vulnerability Type

Missing Best Practices

#### Severity

Low

#### Description

Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log—a special data structure in the blockchain. These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions.

The contract was found to be missing these events on certain critical functions which would make it difficult or impossible to track these transactions off-chain.

#### Affected Code

The following functions were affected -

- ContractV2.sol - addPaymentMode() and updatePaymentMode()

```
function addPaymentMode(address creatorAddress,string memory
PaymentMethodName,address PaymentMethodAddress,bool isActive,bool
isToken,uint feePercentage,bool isPublic) public
{}

function updatePaymentMode(uint index, address creatorAddress, bool
isActive, uint feePercentage, bool isPublic) public
{}
```

## **Impacts**

Events are used to track the transactions off-chain and missing these events on critical functions makes it difficult to audit these logs if they're needed at a later stage.

## **Remediation**

Consider emitting events for the functions mentioned above. It is also recommended to have the addresses indexed.

## **Retest:**

Events have been added to both functions.

<https://github.com/Akestor/Digital-Invoice/blob/909a8eb9b2932244c7508fb2b5891f265d656c84/Contracts/ContractV2.sol#L488>

<https://github.com/Akestor/Digital-Invoice/blob/909a8eb9b2932244c7508fb2b5891f265d656c84/Contracts/ContractV2.sol#L502>

## Bug ID#13 [Fixed]

### Private Invoice's Information Disclosure

#### Vulnerability Type

Information Disclosure

#### Severity

Medium

#### Description

The contract InvoiceNFT implements an enum called "PRIVACY" during invoice creation that keeps track if an invoice is kept private from other users. This is being used along with a modifier "authorizedAccessForBoth()" which is validating the same when getting the invoice. On the blockchain, nothing is private and can be obtained by directly fetching data from the storage slots. Therefore, this approach can be bypassed.

Reference:

[https://docs.soliditylang.org/en/v0.8.17/internals/layout\\_in\\_storage.html#mappings-and-dynamic-arrays](https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_storage.html#mappings-and-dynamic-arrays)

#### Affected Code

The following functions were affected -

- ContractV2.sol

```
function getInvoice(uint256 _tokenId)
    public
    view
    authorizedAccessForBoth(_tokenId)
    returns (INVOICE memory)
{
    require(_exists(_tokenId), "ERROR 114");
    return allInvoices[_tokenId];
}
```

**Impacts**

This current implementation is vulnerable to invoice information disclosure if the user decides to keep their invoices private.

**Remediation**

The feature needs to be removed as this cannot be fixed.

**Retest:**

This issue cannot be fixed and hence it won't be advertised as a privacy feature anymore.

## Bug ID#14 [Fixed]

### Incorrect Invoice Price Calculation

#### Vulnerability Type

Business Logic

#### Severity

High

#### Description

The contract ContractV2.sol has a function to create invoices in which it is calculating the invoice cut “\_cut” using the following calculations:

```
uint256 _cut = (_amount * (mode.feePercentage + privacyFee)) / 100;  
OR  
uint256 _cut = (_amount * mode.feePercentage) / 100;
```

During the invoice’s payment, instead of this cut being deducted from the payer’s balance, the function was not deducting that amount. Instead, the payer had to pay that much less amount when paying for the invoice.

```
bool success = inv.paymentMode.PaymentMode.transferFrom(  
    msg.sender,  
    inv.from,  
    inv.amount - inv.cut  
);
```

#### Attack Scenario

- Alice creates an invoice for 500 tokens for Bob using a payment mode with a fee percentage of 5%.
- Ideally, Bob should pay 500 + the amount calculated after incurring a 5% fee.
- But when this payment was initiated by Bob, he only paid 475.
- This rendered the fee percentage for the payment mode useless.

### **Affected Code**

<https://github.com/Akestor/Digital-Invoice/blob/9777cb9e61212c2d142812f69adc0b56bf983a4f/Contracts/ContractV2.sol#L215>

### **Impacts**

This vulnerability exploits the improper invoice amount calculation due to missing accounting for the fee percentage in the payment mode to pay less than the required amount for the invoice.

### **Remediation**

It is recommended to calculate the fee properly and instead of deducting less amount from the payer, the fee amount should be added to their transaction.

### **Retest:**

The price calculation has been updated to fix the issue.

## 6. Disclosure

---

The Reports provided by CredShields is not an endorsement or condemnation of any specific project or team and do not guarantee the security of any specific project. The contents of this report are not intended to be used to make decisions about buying or selling tokens, products, services, or any other assets and should not be interpreted as such.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. CredShields does not provide any warranty or representation about the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business. The report is not intended to be used as investment advice and should not be relied upon as such.

CredShields Audit team is not responsible for any decisions or actions taken by any third party based on the report.