CredShields

Smart Contract Audit

Aug 8th, 2023 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Daikoku between July 12th, 2023, and July 18th, 2023. And a retest was performed on Aug 4th, 2023.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Daikoku

Table of Contents

CRED SHiELDS

1. Executive Summary

Daikoku engaged CredShields to perform a smart contract audit from July 12th, 2023, to July 18th, 2023. During this timeframe, Fourteen (14) vulnerabilities were identified. **A retest was performed on Aug 4th, 2023, 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 "Daikoku" 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
Smart Contract	0	0	1	1	2	1	**5**
	0	**0**	**1**	**1**	**2**	**1**	**5**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in Smart Contract's scope during the testing window while abiding by the policies set forth by 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 Daikoku'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 Daikoku can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Daikoku can future-proof its security posture and protect its assets.

2. Methodology

Daikoku engaged CredShields to perform a Daikoku 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 July 12th, 2023, to July 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
Daikoku Smart Contracts

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

Daikoku 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

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, Fourteen (14) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Floating Pragma	Low	Floating Pragma (SWC-103)
Missing Price Feed Validation	Medium	Input Validation
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	Missing best practices
Missing Zero Address Validations	Low	Missing Input Validation
Chainlink Oracle Min/Max price validation	Medium	Input Validation
Hardcoded USDC Price	Medium	Input Validation
Users Token Lockout	Medium	Business Logic

Gas Optimization in Require Statements	Gas	Gas Optimization
Variables should be Immutable	Gas	Gas Optimization
Functions should be declared External	Gas	Gas Optimization
Unused Variables	Informational	Dead Code
Missing NatSpec Comments	Informational	Missing best practices
Gas Optimization in Increments	Gas	Gas optimization
Public Constants can be Private	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	The issue persists in versions before v0.8.X.
SWC-102	Outdated Compiler Version	Not Vulnerable	Version 0^.8.0 and above is used
SWC-103	Floating Pragma	Not Vulnerable	Contract uses floating pragma
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	Not Vulnerable	No such scenario was found
SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

4. Remediation Status

Daikoku is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Aug 4th, 2023, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Floating Pragma	Low	**Fixed [04/08/2023]**
Missing Price Feed Validation	Medium	**Fixed [04/08/2023]**
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	**Fixed [04/08/2023]**
Missing Zero Address Validations	Low	**Fixed [04/08/2023]**
Chainlink Oracle Min/Max price validation	Medium	**Fixed [04/08/2023]**
Hardcoded USDC Price	Medium	**Fixed [04/08/2023]**
Users Token Lockout	Medium	**Fixed [04/08/2023]**
Gas Optimization in Require Statements	Gas	**Fixed [04/08/2023]**

Variables should be Immutable	Gas	**Fixed [04/08/2023]**
Functions should be declared External	Gas	**Fixed [04/08/2023]**
Unused Variables	Informational	**Fixed [04/08/2023]**
Missing NatSpec Comments	Informational	**Fixed [04/08/2023]**
Gas Optimization in Increments	Gas	**Fixed [04/08/2023]**
Public Constants can be Private	Gas	**Fixed [04/08/2023]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [Fixed]

Floating Pragma

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., **^0.8.17**.
This allows the contracts to be compiled with all the solidity compiler versions above **0.8.17**.
The following contracts were found to be affected -

Affected Code

- Fundraiser
- TaxCollector
- DaikokuDAO

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.

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

Retest:

The pragma has been fixed and hardcoded to 0.8.17.

Bug ID#2 [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

- Fundraiser.sol - Line 204

```
(,int price,,,) = priceFeed.latestRoundData();
// Convert the price to a uint256 and the ETH amount to USDC
uint256 usdPrice = uint256(price);
uint256 usdcAmount6d = msg.value * usdPrice / 1e20; // 18 + 8 - *20* = 6 decimals

    // console.log("RECEIVED ETH %s", msg.value);
```

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. Here's a sample implementation:

```
(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

This has been fixed. Validations are implemented .

```
(uint80 roundID, int price,, uint256 timestamp, uint80 answeredInRound) = priceFeedEth.latestRoundData();
require(price >= 100*1e8 && price <= 200000*1e8, "ETH price out of bounds"); // Price of ETH must be between 100 USDC and 200k USD
require(answeredInRound >= roundID, "Stale price");
require(timestamp != 0, "Round not complete");
```

Bug ID#3 [Fixed]

Use safeTransfer/safeTransferFrom instead of transfer/transferFrom

Vulnerability Type
Missing best practices

Severity
Low

Description
The transfer() and transferFrom() method is used instead of safeTransfer() and safeTransferFrom(), presumably to save gas however OpenZeppelin's documentation discourages the use of transferFrom(), use safeTransferFrom() whenever possible because safeTransferFrom auto-handles boolean return values whenever there's an error.

Affected Code

- Fundraiser.sol - Line 221
- Fundraiser.sol - Line 285

```
usdc.transferFrom(msg.sender, address(this), usdcAmount6d);
usdc.transfer(treasury, usdcBalance);
```

Impacts
Using safeTransferFrom has the following benefits -

- It checks the boolean return values of ERC20 operations and reverts the transaction if they fail,
- at the same time allowing you to support some non-standard ERC20 tokens that don't have boolean return values.

- It additionally provides helpers to increase or decrease an allowance, to mitigate an attack possible with vanilla approve.

Remediation
Consider using safeTransfer() and safeTransferFrom() instead of transfer() and transferFrom().

Retest
Transfer calls have been updated to safeTransfer/safeTransferFrom.

Bug ID #4 [Fixed]

Missing Zero Address Validations

Vulnerability Type
Missing Input Validation

Severity
Low

Description:
The contracts were found to be setting new addresses without proper validations for zero addresses.
Address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burned forever.
Depending on the logic of the contract, this could prove fatal and the users or the contracts could lose their funds, or the ownership of the contract could be lost forever.

Affected Variables and Line Numbers

- Fundraiser.sol - Line 68-72 - DaikokuDAO _token, IERC20 _usdc, address _chainlinkContract, address _treasury, address _management1, address _management2

Impacts
If address type parameters do not include a zero-address check, contract functionality may become unavailable or tokens may be burned permanently.

Remediation
Add a zero address validation to all the functions where addresses are being set.

Retest
Zero address validation has been added.

Bug ID#5 [Fixed]

Chainlink Oracle Min/Max price 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.
Chainlink aggregators have a built in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value, the price of the oracle will continue to return the minPrice instead of the actual price of the asset.

Vulnerable Code

- Fundraiser.sol - Line 199-212

```
function depositETH() public payable {
        require(msg.value > 0, "Must send ETH");
        Fundraise storage fundraise = getActiveInternal();

    // Get the latest round data from the price feed
    (,int price,,,) = priceFeed.latestRoundData();
    // Convert the price to a uint256 and the ETH amount to USDC
    uint256 usdPrice = uint256(price);
    uint256 usdcAmount6d = msg.value * usdPrice / 1e20; // 18 + 8 - *20* = 6 decimals

        // console.log("RECEIVED ETH %s", msg.value);

        allocateDeposit(fundraise, usdcAmount6d);
```

```
}
```

Impacts

This would allow user to store their allocations with the asset but at the wrong price.

Remediation

The contract should check the returned answer/price against the minPrice/maxPrice and revert if the answer is outside of the bounds.

```
if (answer >= maxPrice or answer <= minPrice) revert(); // eg
```

Retest

```
require(price >= 100*1e8 && price <= 200000*1e8, "ETH price out of bounds");
```

Bug ID#6 [Fixed]

Hardcoded USDC Price

Vulnerability Type
Input Validation

Severity
Medium

Description
The Fundraiser contract has a function to allow users to deposit USDC directly into the contract using the depositUSDC().
This transfers the USDC from the user's wallet into the contract and calculates their allocation based on the assumed value that 1USDC will always be equal to 1 USD which is not always the case.

Vulnerable Code

- Fundraiser.sol - Line 214-224

Impacts
If the price of the asset fluctuates, it would lead to loss/gain of tokens based on the incorrect calculation.

Remediation
It is recommended to use the USDC/USD Chainlink oracle to fetch the price similar to what is done in the depositETH() function. Ensure that the validations are in place.

Retest
Chainlink Oracle has been added to fetch the price.

```
(uint80 roundID, int price,, uint256 timestamp, uint80 answeredInRound)
= priceFeedUsdc.latestRoundData();
```

Bug ID#7 [Fixed]

Users Token Lockout

Vulnerability Type
Business Logic

Severity
Medium

Description
The fundraiser contract allows users to deposit ETH/USDC into the contract after a fundraiser starts. They are only allowed to withdraw their allocations when the fundraiser ends.
However, in case the user forgets to claim their allocation, the tokens might get locked out if the owner starts another fundraiser.

Vulnerable Code
- Fundraiser.sol - Line 261-272

Impacts
This vulnerability could lock the user out of their allocated tokens if the owner starts another fundraise before allowing the user to claim their tokens.

Remediation
It is recommended to have a limit on the maximum number of fundraiser rounds and also the maximum time period for which a fundraiser last to ensure the user that they can withdraw their claim amount.

Retest
Fundraiser duration has been capped at >= 1 hour and <= 2 weeks and the maximum number of fundraisers has been capped at 3.

Bug ID#8 [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. Once such example is given below:

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

Remediation
It is recommended to go through all the **require()** statements present in the contract and shorten the strings passed inside them 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
This has been fixed. The strings inside the require statements have been shortened to less than 32 characters.

Bug ID #9 [Fixed]

Variables should be Immutable

Vulnerability Type
Gas Optimization

Severity
Gas

Description:
Declaring state variables that are not updated following deployment as immutable can save gas costs in smart contract deployments and function executions. Immutable state variables are those that cannot be changed once they are initialized, and their values are set permanently.

By declaring state variables as immutable, the compiler can optimize their storage in a way that reduces gas costs. Specifically, the compiler can store the value directly in the bytecode of the contract, rather than in storage, which is a more expensive operation.

Affected Code:

- TaxCollector - DaikokuDAO public token
- Fundraiser - DaikokuDAO public token, IERC20 public usdc, AggregatorV3Interface internal priceFeed

Impacts:
Gas usage is increased if the variables that are not updated outside of the constructor are not set as immutable.

Remediation:
An "`immutable`" attribute should be added in the parameters that are never updated outside of the constructor to save the gas.

Retest

The variables that are not updated anywhere outside the constructor have been updated as immutable.

Bug ID#10 [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 the public visibility useless.

Affected Code
The following functions were affected -

- DaikokuDAO.addUntaxedAddresses()
- DaikokuDAO.removeUntaxedAddresses()
- Fundraiser.depositETH()
- Fundraiser.claim()
- Fundraiser.depositUSDC()

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 functions that are not called from inside the contracts have been updated as external.

Bug ID#11 [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.

Affected Code

- Fundraiser - uint256 public constant CLIFF_DURATION = 365 days;

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
It is recommended to remove the variable if it is not supposed to be used.

Retest:
The variable is now being used in updating the cliffTime when starting the fundraiser.

Bug ID#12 [Fixed]

Missing NatSpec Comments

Vulnerability Type
Missing best practices

Severity
Informational

Description
Solidity contracts use a special form of comments to document code. This special form is named the Ethereum Natural Language Specification Format (NatSpec).
The document is divided into descriptions for developers and end-users along with the title and the author.
The contracts were missing NatSpec comments in the code which makes it difficult for the auditors and future developers to understand the code.

Impacts
Missing NatSpec comments and documentation about a library or a contract affect the audit and future development of the smart contracts.

Remediation
Add necessary NatSpec comments inside the library along with documentation specifying what it's for and how it's implemented.

Retest
Comments have been added.

Bug ID#13 [Fixed]

Gas Optimization in Increments

Vulnerability Type
Gas optimization

Severity
Gas optimization

Description
The contract uses two for loops**,** which use 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
- DaikokuDAO.addUntaxedAddresses()
- DaikokuDAO.removeUntaxedAddresses()

Impacts
Using **i++** instead of **++i** costs the contract deployment around 600 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 saves some gas.

Retest
Increments have been updated to ++i to optimize gas.

Bug ID #14 [Fixed]

Public Constants can be Private

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Public constant variables cost more gas because the EVM automatically creates getter functions for them and adds entries to the method ID table. The values can be read from the source code instead.

Affected Code

- DaikokuDAO.MAX_SUPPLY
- Fundraiser.DECIMALS, CLIFF_DURATION, WITHDRAWAL_PERIOD_DURATION,

Impacts
Public constants are more costly due to the default getter functions created for them, increasing the overall gas cost.

Remediation
If reading the values for the constants is not necessary, consider changing the public visibility to private.

Retest
The variables have been updated as private to save gas.

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.