CredShields

Smart Contract Audit

Jan 12th, 2023 • CONFIDENTIAL

Description

This document details the process and result of the CGT smart contract audit performed by CredShields Technologies PTE. LTD. on behalf of Coin Gabbar between Jan 9th, 2023, and Jan 11th, 2023. And a retest was performed on Jan 11th, 2023.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Coin Gabbar

Table of Contents

1. Executive Summary

Coin Gabbar engaged CredShields to perform a smart contract audit from Jan 9th, 2023, to Jan 11th, 2023. During this timeframe, Five (5) vulnerabilities were identified. **A retest was performed on Jan 11th, 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 "Coin Gabbar" 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
CGT smart contract	0	0	0	1	3	1	**5**
	0	**0**	**0**	**1**	**3**	**1**	**5**

Table: Vulnerabilities Per Asset in Scope

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

2. Methodology

Coin Gabbar engaged CredShields to perform a Coin Gabbar 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 Jan 9th, 2023, to Jan 11th, 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
https://bscscan.com/token/0xf3744f2a8b2b26896d55ad3f6d06a0382bc00a19#code **https://bscscan.com/address/0x74336d165ee70fc592067da976ff4b15596ae2d1#code**

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

Coin Gabbar 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, Five (5) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Floating Pragma and Inconsistent Pragma versions	Low	Floating Pragma (SWC-103)
Large Number Literals	Gas	Gas & Missing Best Practices
Transfer Hook Optimization	Informational	Code Optimization
Outdated OpenZeppelin Contracts	Informational	Components with Known Vulnerabilities
Typo in comments	Informational	Missing best practices

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	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 such case was found
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	Redundant codes found
SWC-136	[Unencrypted Private Data On-Chain]	Not Vulnerable	No such scenario was found

4. Remediation Status

Coin Gabbar is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Jan 11th, 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 and Inconsistent Pragma versions	Low	**Won't fix [12/1/2023]**
Large Number Literals	Gas	**Won't fix [12/1/2023]**
Transfer Hook Optimization	Informational	**Won't fix [12/1/2023]**
Outdated OpenZeppelin Contracts	Informational	**Won't fix [12/1/2023]**
Typo in comments	Informational	**Won't fix [12/1/2023]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [Won't Fix]

Floating Pragma and Inconsistent 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., **^0.8.9, ^0.8.0,** and **^0.8.2.** 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 - Address.sol
- ^0.8.0 - BeaconProxy.sol
- ^0.8.0 - Context.sol
- ^0.8.0 - ERC1967Proxy.sol
- ^0.8.0 - IBeacon.sol
- ^0.8.0 - import.sol
- ^0.8.0 - Ownable.sol
- ^0.8.0 - Proxy.sol
- ^0.8.0 - ProxyAdmin.sol
- ^0.8.0 - StorageSlot.sol

- ^0.8.0 - TransparentUpgradeableProxy.sol
- ^0.8.0 - UpgradeableBeacon.sol
- ^0.8.9 - CGToken.sol
- ^0.8.2 - ERC1967Upgrade.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 issue is not major and does not affect the security of the token hence it can be ignored. The CredShields team agrees to the fact.

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

- CGToken.sol - Line [1873](#)

```
function initialize() public initializer {
    __ERC20_init("Coin Gabbar Token", "CGT");
    __ERC20Burnable_init();
    __AccessControl_init();

    _mint(msg.sender, 1000000000000 * 10**decimals());
    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
}
```

Impacts

Having a large number 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
It is recommended to use "**1e30**" in this case in place of **1000000000000 * 10**decimals()**.

Retest:

The issue is not major and does not affect the security of the token. It just saves more gas hence it can be ignored. The CredShields team agrees with the fact.

Bug ID#3 [Won't Fix]

Transfer Hook Optimization

Vulnerability Type

Code Optimization

Severity

Informational

Description

The **_transfer** function is being overridden in the main CGToken contract to add the validations for address blacklisting.
The ERC20 OpenZeppelin contract also introduces a predefined hook called **_beforeTokenTransfer** which is called before each token transfer that is meant exactly for this.

Affected Code

- CGToken.sol - Line 1897

```
function _transfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    require(
        !hasRole(BLACKLISTED_ROLE, to),
        "The receiver wallet has been blacklisted by the Admin"
    );
    require(
        !hasRole(BLACKLISTED_ROLE, msg.sender),
        "The sender wallet has been blacklisted by the Admin"
    );
```

```
        return super._transfer(from, to, amount);
    }
```

Impacts

This does not have a security impact because the _transfer code is working as it should without any complications. However, it may affect the gas cost.

Remediation

Instead of overriding **_transfer**, it is recommended to override **_beforeTokenTransfer**.

Retest:

If the function stays it won't have any impact and hence the team decided to leave it as it is.

Bug ID#4 [Won't Fix]

Outdated OpenZeppelin Contracts

Vulnerability Type

Components with Known Vulnerabilities

Severity

Informational

Description

The OpenZeppelin modules used in the code are found to be outdated. These libraries and contracts are affected by multiple CVEs and exploits that are publicly available.
One such case was found in the ERC1967Upgrade.sol contract. This defines a function called "_upgradeToAndCallSecure()" which introduces a bug in the case of UUPS proxies when the logic contract is left uninitialized. However, this was not the case in this contract.

Affected Code

- ERC1967Upgrade.sol - Line 76

```
function _upgradeToAndCallSecure(
    address newImplementation,
    bytes memory data,
    bool forceCall
) internal {
    address oldImplementation = _getImplementation();

    // Initial upgrade and setup call
    _setImplementation(newImplementation);
    if (data.length > 0 || forceCall) {
        Address.functionDelegateCall(newImplementation, data);
    }
```

```
        // Perform rollback test if not already in progress
        StorageSlot.BooleanSlot storage rollbackTesting = StorageSlot
            .getBooleanSlot(_ROLLBACK_SLOT);
        if (!rollbackTesting.value) {
            // Trigger rollback using upgradeTo from the new
implementation
            rollbackTesting.value = true;
            Address.functionDelegateCall(
                newImplementation,
                abi.encodeWithSignature("upgradeTo(address)",
oldImplementation)
            );
            rollbackTesting.value = false;
            // Check rollback was effective
            require(
                oldImplementation == _getImplementation(),
                "ERC1967Upgrade: upgrade breaks further upgrades"
            );
            // Finally reset to the new implementation and log the
upgrade
            _setImplementation(newImplementation);
            emit Upgraded(newImplementation);
        }
    }
```

Impacts

The above was just one of the examples of a vulnerable code. But fortunately, it was not affecting the CGToken implementation. This does not mean that the older version of contracts is secure as they are more prone to vulnerabilities discovered in the future.

Remediation

It is recommended to update all the contracts to their latest and patched versions to prevent any present or future vulnerabilities.

Retest:

Although there is an updated version of the code however the current older implementation doesn’t cause an exploitable scenario and hence the team decided to leave it as it is.

Bug ID#5 [Won't Fix]

Typo in Comments

Vulnerability Type
Missing best practices

Severity
Informational

Description
The contracts were found to be using the wrong spelling for certain words in the comments. These may cause confusion for future auditors or developers when reading and understanding the code.

Affected Code

- Proxy.sol - Line 19, 44, 52, 79 eg: internall, overriden

Impacts
Typo in comments may cause issues when a developer or auditor is trying to understand the code as the comments are the main source of documentation in the initial audit stages.

Remediation
It is recommended to correct the spellings of the words highlighted above.

Retest:
The team decided not to fix it as it won't cause any exploitable scenario.

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.