CredShields
Smart Contract Audit

Nov 10th, 2022 • CONFIDENTIAL

Description

This document details the process and result of the DeVo Protocol DeVo Token Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of DeVo Protocol between Nov 5th, 2022, and Nov 8th, 2022. And a retest was performed on 12th Nov 2022.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

DeVo Protocol

Table of Contents

1. Executive Summary

DeVo Protocol engaged CredShields to perform a smart contract audit from Nov 5th, 2022, to Nov 8th, 2022. During this timeframe, Five (5) vulnerabilities were identified. **A retest was performed on 12th Nov 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 "DeVo Protocol" 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
DeVo Token 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 DeVo Protocol's scope during the testing window while abiding by the policies set forth by DeVo Protocol'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 DeVo Protocol'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 DeVo Protocol introduces new features or refactors the code.

Reviewing the remaining resolved reports for a root cause analysis can further educate DeVo Protocol'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 DeVo Protocol assets.

2. Methodology

DeVo Protocol engaged CredShields to perform a DeVo Protocol 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 Nov 5th, 2022, to Nov 8th, 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
DeVo_Token.sol

Table: List of Files in Scope

2.1.2 Documentation

N/A - 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](https://swcregistry.io/) 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

DeVo Protocol 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, Five (5) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Hardcoded Static Address	Informational	Missing Best Practises
Use of SafeMath	Gas	Gas Optimization
Large Number Literals	Informational	Missing Best Practises
Frontrunning	Medium	Frontrunning
Missing Input Validation	Low	Input Validation

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

DeVo Protocol is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on 10th Nov 2022, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Hardcoded Static Address	Informational	**Fixed [12/11/2022]**
Use of SafeMath	Gas	**Fixed [12/11/2022]**
Large Number Literals	Informational	**Fixed [12/11/2022]**
Frontrunning	Medium	**Fixed [12/11/2022]**
Missing Input Validation	Low	**Fixed [12/11/2022]**
Hardcoded Static Address	Informational	**Fixed [12/11/2022]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [Fixed]

Hardcoded Static Address

Vulnerability Type
Missing Best Practises

Severity
Informational

Description
The contract “SPOCToken” was found to be using hardcoded addresses on Line 104 and 105. This could have been optimized using dynamic address update techniques along with proper access control to aid in address upgrade at a later stage.

Affected Code

- SPOCToken - [Line 104 & 105]

```
  constructor() {
      symbol = "DVO";
      name = "DEVO Ethereum Token";
      decimals = 18;
      _totalSupply = 1500000000000000000000000000;
      //Need to update owner wallet
0x4483Cc90a7c6bf767b3c240c3465987D5Cc95E64
      balances[0x4483Cc90a7c6bf767b3c240c3465987D5Cc95E64] =
_totalSupply;
      emit Transfer(address(0),
```

```
0x4483Cc90a7c6bf767b3c240c3465987D5Cc95E64, _totalSupply);
    }
```

Impacts

Hardcoding address variables in the contract make it difficult for it to be modified at a later stage in the contract as everything will need to be deployed again at a different address if there's a code upgrade.

Remediation

It is recommended to create dynamic functions to address upgrades so that it becomes easier for developers to make changes at a later stage if necessary.

The said function should have proper access controls to ensure only administrators can call that function using access control modifiers.

There should also be a zero address validation in the function to ensure inconsistencies are not introduced.

If the address is supposed to be hardcoded, it is advisable to make it a constant if its value is not getting updated.

Retest:

-

Bug ID#2 [Fixed]

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, rendering SafeMath unnecessary.

Affected Code:

- SPOCToken - [Line 85]

```
...
contract SPOCToken is ERC20Interface, Owned, SafeMath {
...
```

Impacts:
This increases the gas usage of the contract.

Remediation:
We do not recommend using the 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 above 0.8.0+ automatically checks for overflows and underflows.

Retest:
SafeMath has been removed.

Bug ID#3 [Fixed]

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

- SPOCToken - [Line 102]

```
constructor() {
    symbol = "DVO";
    name = "DEVO Ethereum Token";
    decimals = 18;
    _totalSupply = 1500000000000000000000000000;
```

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. This can be rewritten as **15e26**.

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

Retest:

Now, the scientific notion is in use.

Bug ID#4 [Fixed]

Approve Fruntrunning Attack

Vulnerability Type

Frontrunning

Severity

Medium

Description

The contract **SPOCToken** is defining an “approve” function on line 143, 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 - a victim and an attacker and the attacker must be monitoring the transactions to frontrun approval calls.

Vulnerable Code

- SPOCToken - [Line 143]

```
    function approve(address spender, uint tokens) public override
returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }
```

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 has been removed and hence no more vulnerable.

Bug ID#5 [Fixed]

Missing Input Validation

Vulnerability Type

Input validation

Severity

Low

Description:

Input validation is a frequently-used technique for checking potentially dangerous inputs in order to ensure that the inputs are safe for processing within the code, or when communicating with other components.

When the smart contract does not validate the inputs properly, it may introduce a range of vulnerabilities.

The contract did not implement any input validation when transferring amounts inside “transfer()” and “transferFrom()” functions.

Vulnerable Code:

- SPOCToken - [Lines 130 and 159]

```
   function transfer(address to, uint tokens) public override returns
(bool success) {
       balances[msg.sender] = safeSub(balances[msg.sender], tokens);
       balances[to] = safeAdd(balances[to], tokens);
       emit Transfer(msg.sender, to, tokens);
       return true;
   }
...

...
```

```
    function transferFrom(address from, address to, uint tokens) public
override returns (bool success) {
        balances[from] = safeSub(balances[from], tokens);
        allowed[from][msg.sender] = safeSub(allowed[from][msg.sender],
tokens);
        balances[to] = safeAdd(balances[to], tokens);
        emit Transfer(from, to, tokens);
        return true;
    }
```

Impacts:

If the amount is set to 0 by mistake or due to any errors, it will just cost unnecessary gas.

Remediation:

Use a `require()` input validation in the amount parameter to make sure it's never set to 0.

Retest:

The function has been removed from the contract.

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.