CredShields
Smart Contract Audit

Nov 10st, 2022 • CONFIDENTIAL

Description

This document details the process and result of the FoundersLab Token smart contract audit performed by CredShields Technologies PTE. LTD. on behalf of FoundersLab between Oct 21st, 2022, and Oct 30th, 2022. And a retest was performed on 8th Nov 2022.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

FoundersLab

Table of Contents

1. Executive Summary

FoundersLab engaged CredShields to perform a smart contract audit from Oct 21, 2022, to Oct 30th, 2022. During this timeframe, twelve (12) vulnerabilities were identified. **A retest was performed on 8th Nov 2022, and all the bugs have been addressed.**

During the audit, four (4) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "FoundersLab" 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
FoundersLab	2	2	0	3	3	2	**12**
	2	**2**	**0**	**3**	**3**	**2**	**12**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in FoundersLab's scope during the testing window while abiding by the policies set forth by FoundersLab'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 FoundersLab'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 FoundersLab introduces new features or refactors the code.

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

2. Methodology

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

2.1 Preparation phase

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

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

A testing window from Oct 21st, 2022, to Oct 30th, 2022, was agreed upon during the preparation phase.

2.1.1 Scope

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

IN SCOPE ASSETS
https://github.com/Founders-Lab/FL-New-Smart-Contract

Table: List of Files in Scope

2.1.2 Documentation

https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/main/README.md

2.1.3 Audit Goals

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

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

- **Security** - Identifying security-related issues within each contract and the system of contracts.

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

2.2 Retesting phase

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

2.3 Vulnerability classification and severity

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

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

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

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

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

Overall, the categories can be defined as described below -

1. **Informational**

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

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

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

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

2. Low

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

3. Medium

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

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

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

4. **High**

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

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

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

   They should be fixed immediately.

5. **Critical**

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

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

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

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

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

2.4 CredShields staff

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

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

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

3. Findings

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

3.1 Findings Overview

3.1.1 Vulnerability Summary

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

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Floating Pragma	Low	Floating Pragma (SWC-103)
Large Number Literals	Informational	Missing Best Practices
Hardcoded Static Address	Informational	Missing Best Practices
Gas Optimization in Require Statements	Gas	Gas Optimization
Missing Multiple Zero Address Validations	Low	Missing Input Validation
Missing Access Control in “updateXP()”	High	Missing Access Control
Missing Access Control in “updateUSDCAmount()”	Critical	Missing Access Control

Missing Input Validation	Low	Input validation
Functions should be declared External	Gas	Gas Optimization
Missing Reentrancy Protections	High	Reentrancy (SWC-107)
Dead Code	Informative	Code With No Effects - SWC-135
Missing Access Control in Burn Function	Critical	Improper Access Control

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

4. Remediation Status

FoundersLab is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on 30th Oct 2022 and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Floating Pragma	Low	**Fixed [8/11/2022]**
Large Number Literals	Informational	**Fixed [8/11/2022]**
Hardcoded Static Address	Informational	**Fixed [8/11/2022]**
Gas Optimization in Require Statements	Gas	**Fixed [8/11/2022]**
Missing Multiple Zero Address Validations	Low	**Fixed [8/11/2022]**
Missing Access Control in “updateXP()”	High	**Fixed [8/11/2022]**
Missing Access Control in “updateUSDCAmount()”	Critical	**Fixed [8/11/2022]**
Missing Input Validation	Low	**Fixed**

		[8/11/2022]
Functions should be declared External	Gas	**Fixed [8/11/2022]**
Missing Reentrancy Protections	High	**Fixed [8/11/2022]**
Dead Code	Informative	**Fixed [8/11/2022]**
Missing Access Control in Burn Function	Critical	**Fixed [8/11/2022]**

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 allowed floating or unlocked pragma to be used, i.e., >=0.7.0<0.9.0 and ^0.8.3.
This allows the contracts to be compiled with all the solidity compiler versions above **0.7.0**.
The following contracts were found to be affected -

Affected Code

- Pragma version ^0.8.3 - (contracts/FNDR.sol#2)
- Pragma version >=0.7.0<0.9.0 - (contracts/Profile.sol#2)
- Pragma version >=0.7.0<0.9.0 - (contracts/ProfileBeacon.sol#2)
- Pragma version >=0.7.0<0.9.0 - (contracts/ProfileFactory.sol#2)
- Pragma version >=0.7.0<0.9.0 - (contracts/Project.sol#2)
- Pragma version >=0.7.0<0.9.0 - (contracts/ProjectBeacon.sol#2)
- Pragma version >=0.7.0<0.9.0 - (contracts/ProjectFactory.sol#2)
- Pragma version >=0.7.0<0.9.0 - (contracts/Task.sol#3)
- Pragma version >=0.7.0<0.9.0 - (contracts/TaskBeacon.sol#2)
- Pragma version >=0.7.0<0.9.0 - (contracts/TaskFactory.sol#3)
- Pragma version ^0.8.3 - (contracts/USDC.sol#2)

- Pragma version >=0.7.0<0.9.0 - (contracts/interfaces/IFNDRInterface.sol#3)
- Pragma version >=0.7.0<0.9.0 - (contracts/interfaces/IProfileFactoryInterface.sol#3)
- Pragma version >=0.7.0<0.9.0 - (contracts/interfaces/IProfileInterface.sol#3)
- Pragma version >=0.7.0<0.9.0 - (contracts/interfaces/IProjectFactoryInterface.sol#3)
- Pragma version >=0.7.0<0.9.0 - (contracts/interfaces/IProjectInterface.sol#3)
- Pragma version >=0.7.0<0.9.0 - (contracts/interfaces/ITaskFactoryInterface.sol#3)
- Pragma version >=0.7.0<0.9.0 - (contracts/interfaces/ITaskInterface.sol#3)

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.

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:

Instead of floating pragma now strict pragma is in use 0.8.9.

Bug ID#2 [Fixed]

Large Number Literals

Vulnerability Type
Missing Best Practices

Severity
Informational

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

- contracts/FNDR.sol#8-10
- contracts/USDC.sol#8-10

```
//contracts/FNDR.sol
   constructor() ERC20("FoundersLab", "FNDR") {
       _mint(_msgSender(), 100000000 * (10**18));
   }

//contracts/USDC.sol
   constructor() ERC20("USDC-FoundersLab", "USDC") {
       _mint(_msgSender(), 100000000 * (10**18));
   }
```

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

Retest:

Now scientific notion is in use.
https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/main/contracts/FNDR.sol#L11

Bug ID#3 [Fixed]

Hardcoded Static Address

Vulnerability Type
Missing Best Practises

Severity
Informational

Description
The contract “Task.sol” was found to be using hardcoded addresses on Line 41-42. A public address variable “FL_wallet” was defined which was using a hardcoded address.
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

- Task.sol 41-42 - link

```
address public FL_wallet =
    address(0x5EE8D1B65942116dA65700849d65cc839E38cC5e);
```

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 make sure only administrators can call that function using access control modifiers.
There should also be a zero address validation in the function to make sure the tokens are not lost.
If the address is supposed to be hardcoded, it is advisable to make it a constant if its value is not getting updated.

Retest:
Hardcoded addresses have been removed and will be added via input.

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

- TaskFactory - 42
- Profile - 51
- Profile - 62
- Task - 126
- Task - 246

```
require(
    projectToTaskAddress[_projectAddress] == address(0),
    "Your project task already created"
);
require(
    projectTaskContracts[_projectTaskAddress],
    "XP can be updated by only task contract"
);
require(
```

```
            projectTaskContracts[_projectTaskAddress],
            "XP can be updated by only task contract"
        );
        require(
            builderProfileAddress != address(0),
            "Builder profile does not exist on chain"
        );
        require(
            idToTask[_taskId].currentState != State.COMPLETE &&
                idToTask[_taskId].currentState != State.CANCELLED,
            "Task already completed or cancelled"
        );
```

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

All possible require statements have been reduced to less than 32 bytes of string.

Bug ID#5 [Fixed]

Missing Multiple Zero Address Validations

Vulnerability Type
Missing Input Validation

Severity
Low

Description:
Multiple Solidity 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

- https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/main/contracts/ProfileFactory.sol#L15-L17

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:
All places where an input address was required has a zero address validation now.

Bug ID#6 [Fixed]

Missing Access Control in “updateXP()”

Vulnerability Type

Missing Access Control

Severity

High

Description:

The contract allows the creation of Profiles using the contract “ProfileFactory.sol” with “createProfile()”. This profile blueprint can be found in the contract “Profile.sol”. This contract has a function that lacks access control - “updateXP()”.

Even though the function is checking using a require statement that it should be called only by the Task contract, the validation is invalid and can be bypassed if an attacker gets the profile address directly by calling the public mapping “ownerToProfileAddress”.

This will give the attacker the profile for the user and they’ll be directly able to call the functions to update XP by specifying the “_projectTaskAddress” which would already be whitelisted, thereby, increasing the parameters infinitely.

Affected Variables and Line Numbers

- https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/main/contracts/Profile.sol#L49-L56

PoC

1. Create a profile by calling the “createProfile()” function.
2. Call the “ownerToProfileAddress” mapping to fetch the address of the profile for the user.
3. Call the “updateXP()” on the profile address, make sure to use the whitelisted “_projectTaskAddress” in the function arguments.

Impacts

This vulnerability allows any external attacker to call the functions to update the profile's XP infinitely.

Remediation

There should be a properly implemented access control modifier on the function that validates that the Task contract's address matches with the "msg.sender" of the transaction, something similar to onlyOwner.

Retest:

A new modifier "checkProjectTask" has been added to the updateXP() function.
https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/main/contracts/Profile.sol#L51-L54

Bug ID#7 [Fixed]

Missing Access Control in “updateUSDCAmount()”

Vulnerability Type

Missing Access Control

Severity

Critical

Description:

The contract allows the creation of Profiles using the contract “ProfileFactory.sol” with “createProfile()”. This profile blueprint can be found in the contract “Profile.sol”. This contract has a function that lacks access controls - “updateUSDCAmount()”.

Even though the function is checking using a require statement that it should be called only by the Task contract, the validation is invalid and can be bypassed if an attacker gets the profile address directly by calling the public mapping “ownerToProfileAddress”.

This will give the attacker the profile for the user and they’ll be directly able to call the functions to update the USDC amount by specifying the “_projectTaskAddress” which would already be whitelisted, thereby, increasing the USDC infinitely.

Affected Variables and Line Numbers

- https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/main/contracts/Profile.sol#L58-L67

PoC

4. Create a profile by calling the “createProfile()” function.
5. Call the “ownerToProfileAddress” mapping to fetch the address of the profile for the user.
6. Call the “updateUSDCAmount()” on the profile address, make sure to use the whitelisted “_projectTaskAddress” in the function arguments.

Impacts

This vulnerability allows any external attacker to call the functions to update the profile's USDC amount infinitely.

Remediation

There should be a properly implemented access control modifier on the function that validates that the Task contract's address matches with the "msg.sender" of the transaction, something similar to onlyOwner.

Retest:

A new modifier "checkProjectTask" has been added to the updateUSDCAmount() function.
https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/main/contracts/Profile.sol#L56-L59

Bug ID#8 [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 setting the "_amount" for in the "fundProjectTask()". It should not be 0.

Vulnerable Code:

- https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/main/contracts/Project.sol#L41-L54

```
function fundProjectTask(
    address _taskFactoryAddress,
    address _projectAddress,
    uint256 _amount
) external onlyOwner {
    ITaskFactoryInterface taskFactory = ITaskFactoryInterface(
        _taskFactoryAddress
    );
    address _projectTaskAddress = taskFactory.getProjectTask(
        _projectAddress
```

```
        );
        uint256 usdcAmount = (_amount * 110) / 100;
        USDCtoken.transfer(_projectTaskAddress, usdcAmount);
    }
```

Impacts:

If the amount is set to 0 by mistake or due to any errors, it may introduce an erroneous logic or condition within the transaction.

Remediation:

Use a `require()` input validation in the function parameters shown above.

Retest:

A require validation has been added for a bounder check value above 0.

https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/main/contracts/Project.sol#L50

Bug ID#9 [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 -

- updateProfileName
- updateProfileTitle
- Profile.withdraw
- ProfileBeacon.update
- createProfile
- Project.withdraw
- depositFunds
- ProjectBeacon.update
- createProject
- cancelTask
- TaskBeacon.update
- createProjectTask

Impacts

Smart Contracts are required to have effective Gas utusage 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:
All affected public functions have been marked as external to save gas.

Bug ID#10 [Fixed]

Missing Reentrancy Protections

Vulnerability Type

Reentrancy (SWC-107)

Severity

High

Description

In a Reentrancy attack, a malicious contract calls back into the calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways.

The smart contract was missing reentrancy protection on the following functions making external calls -

Affected Code

- Task.cancelTask()
- Task.completeTask()
- Task.commitToTask()
- ProfileFactory.createProfile()
- ProjectFactory.createProject()
- TaskFactory.createProjectTask()

Impacts

Lacking reentrancy protection could allow threat actors to abuse the functions and reenter the contract.

This can lead to excessive interactions with the functions and loss of funds and tokens.

Remediation

Add a Reentrancy guard to the function making external calls.

Retest:

Reentrancy Guard has been added to the affected functions.

Bug ID#11[Fixed]

Dead Code

Vulnerability Type
Code With No Effects - SWC-135

Severity
Informative

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

- ITaskInterface.sol

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 interface is not supposed to be used anywhere, it is suggested to remove the code.

Retest:
The dead code has been removed.
https://github.com/Founders-Lab/FL-New-Smart-Contract/commit/f296fe53d9c61aecfb7c1c570639490a3e96e72a

Bug ID#12 [Fixed]

Missing Access Control in Burn Function

Vulnerability Type

Improper Access Control

Severity

Critical

Description

The visibility of the burn function implemented by the FNDR.sol contract is set to external, allowing any user or attacker to call the function to burn tokens from any user account by specifying their address.

Affected Code

- https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/d0962d71b59f4c93cc5d9848a86a0e7679a29a04/contracts/FNDR.sol#L17-L20

```
function burn(address _account, uint256 _amount) external {
    _burn(_account, _amount);
}
```

Impacts

The burn function can be called by anyone to burn the tokens for that account.

Remediation

Add an access control validation to the burn function so that only the contract owners or account owners are allowed to call them.

Retest:

onlyOwner modifier has been added to address the access control issue.

https://github.com/Founders-Lab/FL-New-Smart-Contract/blob/main/contracts/FNDR.sol#L18

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.