CredShields

Smart Contract Audit

June 21st, 2023 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of PePay between June 4th, 2023, and June 9th, 2023, and a retest was performed on June 19th, 2023.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

PePay

Table of Contents

1. Executive Summary **2**
State of Security 3
2. Methodology **4**
2.1 Preparation phase 4
2.1.1 Scope 5
2.1.2 Documentation 5
2.1.3 Audit Goals 5
2.2 Retesting phase 6
2.3 Vulnerability Classification and Severity 6
2.4 CredShields staff 9
3. Findings **10**
3.1 Findings Overview 10
3.1.1 Vulnerability Summary 10
3.1.2 Findings Summary 12
4. Remediation Status **16**
5. Bug Reports **17**
Bug ID #1 [Fixed] 17
Missing State Variable Visibility 17
Bug ID#2 [Fixed] 19
Missing Events in important functions 19
Bug ID#3 [Fixed] 21
Unused Receive/Fallback Functions 21
Bug ID#4 [Fixed] 22
Functions should be declared External 22
Bug ID#5 [Fixed] 23
Unused Imports 23
Bug ID#6 [Won't Fix] 24
Missing NatSpec Comments 24
Bug ID#7 [Fixed] 25
Use safeTransferFrom instead of transferFrom 25
6. Disclosure **26**

1. Executive Summary

PePay engaged CredShields to perform a smart contract audit from June 4th, 2023, to June 9th, 2023. During this timeframe, Seven (7) vulnerabilities were identified. **A retest was performed on June 19th, 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 "PePay" 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	0	2	3	2	**7**
	0	**0**	**0**	**2**	**3**	**2**	**7**

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 PePay 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 PePay'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 PePay can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, PePay can future-proof its security posture and protect its assets.

2. Methodology

PePay engaged CredShields to perform a PePay 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 June 4th, 2023, to June 9th, 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://github.com/pegasussx/pepay-contracts/tree/main/src

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

PePay 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, Eight (8) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Missing State Variable Visibility	Informational	Missing Best Practices
Missing Events in important functions	Low	Missing Best Practices
Unused Receive/Fallback Functions	Low	Missing Best Practices
Functions should be declared External	Gas	Gas Optimization
Unused Imports	Gas	Gas Optimization
Missing NatSpec Comments	Informational	Missing best practices
Use safeTransferFrom instead of transferFrom	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	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

PePay is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on June 19th, 2023, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Missing State Variable Visibility	Informational	**Fixed [19/06/2023]**
Missing Events in important functions	Low	**Fixed [19/06/2023]**
Unused Receive/Fallback Functions	Low	**Fixed [19/06/2023]**
Functions should be declared External	Gas	**Fixed [19/06/2023]**
Unused Imports	Gas	**Fixed [19/06/2023]**
Missing NatSpec Comments	Informational	**Won't Fix**
Use safeTransferFrom instead of transferFrom	Informational	**Fixed [19/06/2023]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID #1 [Fixed]

Missing State Variable Visibility

Vulnerability Type
Missing Best Practices

Severity
Informational

Description
In Solidity, the visibility of state variables is important as it determines how those variables can be accessed and modified by other contracts or functions.
The contract defined state variables that were missing a visibility modifier.

Affected Code

- /src/PEPAYStaking.sol - PEPAYFeeSplitter FEE_ADDRESS

Impacts
If the visibility of a state variable is accidentally left out, it can cause unexpected behavior and security vulnerabilities. For example, if a state variable is supposed to be private and is accidentally declared without any visibility keyword, it will be treated as "internal" by default, which may lead to it being accessible by other contracts or functions outside the intended scope. This can lead to a potential attack vector for malicious actors.

Remediation
Explicitly define visibility for all state variables. These variables can be specified as public, internal, or private.

Retest

State variable visibility has been specified.

https://github.com/pegasussx/pepay-contracts/blob/main/src/PEPAY.sol#L14

Bug ID#2 [Fixed]

Missing Events in important functions

Vulnerability Type
Missing Best Practices

Severity
Low

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

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

Affected Code
The following functions were affected -

- /src/PEPAYFeeSplitter.sol - setTEAMWallet(), setSTAKERWallet(), setOPWallet(), setLPWallet()
- /src/PEPAY.sol - setFeeAddress()
- /src/PEPAYStaking.sol - setFeeAddress()

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

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

Retest

Events have been emitted for the functions specified above.

https://github.com/pegasussx/pepay-contracts/blob/main/src/PEPAYFeeSplitter.sol#L54-L76

https://github.com/pegasussx/pepay-contracts/blob/main/src/PEPAY.sol#L96

https://github.com/pegasussx/pepay-contracts/blob/main/src/PEPAYStaking.sol#L50

Bug ID#3 [Fixed]

Unused Receive/Fallback Functions

Vulnerability Type
Missing Best Practices

Severity
Low

Description
The contract was found to be defining an empty receive function.
It is not recommended to leave them empty unless there's a specific use case such as to receive Ether via an empty receive() function.

Affected Code

- /src/PEPAYFeeSplitter.sol
- /src/PEPAY.sol
- /src/PEPAYStaking.sol

Impacts
Leaving empty fallback and receive functions may represent missing code or validations when functions are called or ETH is sent to the contract.

Remediation
It is recommended to go through the code to make sure these functions are properly implemented and are not missing any validations in the definition.

Retest
Receive and fallback functions have been removed.

Bug ID#4 [Fixed]

Functions should be declared External

Vulnerability Type

Gas Optimization

Severity

Gas

Description

Public functions that are never called by a contract should be declared external in order to conserve gas.

The following functions were declared as public but were not called anywhere in the contract, making public visibility useless.

Affected Code

The following functions were affected -

- /src/PEPAY.sol - setMax()
- /src/PEPAYStaking.sol - setStart()
- /src/PEPAY.sol - deposit()

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

PEPAY.deposit() function has been removed. The other functions have been updated as external.

Bug ID#5 [Fixed]

Unused Imports

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Solidity is a Gas-constrained language. Having unused code or import statements incurs extra gas usage when deploying the contract.
The contract was found to be importing the file ERC20Burnable.sol in PEPAYFeeSplitter.sol which is not used anywhere in that contract.

Affected Code
The following functions were affected -

- /src/PEPAYFeeSplitter.sol - import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";

Impacts
Having unnecessary dead code in the production contract costs extra gas and makes it difficult for auditors and developers.

Remediation
It is recommended to remove the import statement if it's not supposed to be used.

Retest
The dead code has been removed.

Bug ID#6 [Won’t Fix]

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 in the scope were missing these comments.

Impacts:
Without Natspec comments, it can be challenging for other developers to understand the code's intended behavior and purpose. This can lead to errors or bugs in the code, making it difficult to maintain and update the codebase. Additionally, it can make it harder for auditors to evaluate the code for security vulnerabilities, increasing the risk of potential exploits.

Remediation:
Developers should review their codebase and add Natspec comments to all relevant functions, variables, and events. Natspec comments should include a description of the function or event, its parameters, and its return values.

Retest:
Won’t fix.

Bug ID#7 [Fixed]

Use safeTransferFrom instead of transferFrom

Vulnerability Type
Missing best practices

Severity
Informational

Description:
The transferFrom() method is used instead of safeTransferFrom(), presumably to save gas however OpenZeppelin's documentation discourages the use of transferFrom(), use safeTransferFrom() whenever possible.

Affected Code

- [PEPAYStaking.sol]() - initialize()

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 safeTransferFrom instead of transferFrom.

Retest
safeTransferFrom is used now instead of transferFrom.
https://github.com/pegasussx/pepay-contracts/blob/main/src/PEPAYStaking.sol#L71

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.