IMMUNEBYTES

Security Assessment

Smart Contract Audit

04th April 2025

Forkast.gg
v1.0

Introduction

Name	Forkast
Website	https://forkast.gg/
Repository/Source	https://github.com/TournamentDAO/forkast-sc/blob/master/src/cgpc/PlatformCredits.sol
Platform	L1
Network	Ethereum and EVM compatible chains
Language	Solidity
Commit Hash	acaa8ae2fec7aa923ad0120ae1928effffb7a79b
Timeline	20th Mar 2025 - 25th Mar 2025

Table of Content

Executive Summary

This feature audit report covers the security review of a feature implementation for the Forkast's Platform Credits smart contract. The audit commenced between 20th and 25th March 2025, and this report captures the observations from the final day of the ImmuneBytes security team's assessment of the scoped contract.

The contract is designed to manage an internal currency for on-platform trading by allowing users to deposit collateral tokens and mint Platform Credits (PC). While it leverages robust OpenZeppelin libraries and supports multi-token deposits with precise conversion logic, our review identified several key issues. Ambiguous parameter naming, redundant pre-transfer validations, and a centralized withdrawal approval mechanism, without cross-verification of persistent deposit records, pose potential risks of variable severity impacts.

This audit focuses on verifying the integrity of the deposit–withdrawal lifecycle, the clarity and security of parameter handling, and the robustness of role-based access controls. Our evaluation also emphasizes ensuring that the contract's operational parameters align with established business requirements to enhance overall security and user trust in the Forkast Platform Credits system.

Project Overview

The Forkast Platform Credits contract is an ERC20-based token designed exclusively for use within the Forkast trading platform. It enables users to:

- **Deposit multiple supported tokens**: While documentation emphasizes a 1:1 pairing with USDC, the contract supports other tokens by employing a conversion rate mechanism using a private PRECISION constant.
- **Mint and manage Platform Credits (PC)**: PC tokens are minted based on deposited collateral and are intended for trading on the platform.
- **Withdraw collateral**: Withdrawals are subject to operator approval, ensuring that only pre-approved users can reclaim their deposited funds.

Engagement Goals

1. Ensure that the implemented smart contract behavior (e.g., multi-token handling, withdrawal approvals) is consistent with or clearly communicated in the platform's public documentation.
2. Validate that role-based permissions (e.g., MINTER_ROLE, OPERATOR_ROLE, WHITELIST_ROLE) are correctly enforced to prevent unauthorized actions.
3. Assess the robustness of the EIP712 and ECDSA signature verification process, particularly regarding the potential for replay attacks.
4. Confirm that fee parameters, deposit limits, and minimum withdrawal amounts are set within acceptable bounds and that misconfigurations are prevented.
5. Identify and eliminate any redundant or inefficient code paths (such as unnecessary pre-checks for token allowances or balances) to optimize gas usage and reduce complexity.
6. Evaluate the risks and benefits of the centralized withdrawal approval mechanism and its potential impact on user autonomy and trust.

System Architecture

Core Token Logic:

The contract extends OpenZeppelin's ERC20 implementation, managing the minting and burning of Platform Credits. An EIP712 domain separator is used to support typed data signing for withdrawal approvals.

Multi-Token Deposit and Withdrawal:

The contract supports collateral deposits in multiple tokens, using a conversion mechanism based on predefined rates and a PRECISION constant. This design supports tokens with varying decimals, ensuring accurate conversions.

Centralized Withdrawal Approval:

Withdrawal operations are gated by an operator-controlled approval system (via withdrawApproval mapping). Only users approved through approveWithdraw or approveWithdrawWithSig can withdraw collateral.

Fee Management:

Fees on withdrawals are managed through an inherited FeeManager. However, the contract instantiation does not enforce minimum or maximum fee limits, and threshold values (minimumWithdraw and depositLimit) are set without bounds, exposing the system to potential misconfigurations.

Security Checklist

During our comprehensive audit of the Forkast Platform Credit project, we have assessed the following potential vulnerabilities inspired by common issues encountered in Ethereum-based systems:

Access Control Issue	✅
Initialization Vulnerabilities	✅
Reentrancy Attacks	✅
Logic and Calculation Errors	✅
Unintended State Manipulation	✅
ERC20 Compliance	✅
Token Swapping Risks	✅
Denial of Service (DoS)	✅
Ownership Validation	✅
Integer Overflow and Underflow	✅
Error Handling	✅
Frontrunning Risks	✅
Audit-Specific Issues	✅
Merkle Tree and Leaf Nodes	✅

Methodology

ImmuneBytes team has performed thorough testing of the project, starting with analyzing the code design patterns where the smart contracts and provided technical documentation were reviewed to ensure the architecture of the codebase aligns with the business specifications from a bird's eye view. During this phase, the invariants were identified along with execution of the pre-existing test suite to ensure smooth working of the implemented functionality and set up for the auditor's edge case testing in upcoming phases

Our team then performed a formal line-by-line inspection of the Smart Contract to find potential issues like Signature Replay Attacks, Unchecked External Calls, Merkle Tree Exploits, Reentrancy, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the next phase, the team ran curated test cases in order to verify the findings and issues identified during previous phases so as to provide a proof of concept. The team also runs automated tests over the provided code base using a combination of external and in-house-developed tools to identify vulnerabilities and security flaws of static nature.

The code was audited by a team of independent auditors, which included -

- Testing the functionality of the Smart Contracts to determine proper logic has been followed throughout.
- Thorough, line-by-line review of the code, done by a team of security researchers with experience across Blockchain security, cybersecurity, software engineering, game theory, economics and finance.
- Deploying the code on localnet using open source testing frameworks and clients to run live tests.
- Analyzing failing transactions to check whether the Smart Contracts implementation handles bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest secure version.
- Analyzing the security of the on-chain data and components.

Security Review Report

Severity	Open	Acknowledged	Partially Resolved	Resolved	TOTAL
Critical	-	-	-	-	-
High	-	-	-	-	-
Medium	-	1	-	-	1
Low	-	2	-	-	2
Informational	-	1	-	-	1
TOTAL	-	4	-	-	4

Severity Reference

1. Critical
   Issues causing protocol insolvency, unauthorized theft, or irreversible loss or alteration of assets, or governance outcomes.

2. High
   Issues involving theft or freezing of unclaimed yield, royalties, or temporary locking of assets.

3. Medium
   Operational failures, inefficiencies, or exploitations causing delays, excessive gas usage, or disruption without direct loss.

4. Low
   Minor deviations in promised functionality without impact on underlying value.

5. Informational
   Recommendations for code readability, maintainability, or best practices.

Status Reference

1. Open
   The issue has been identified and is pending resolution.

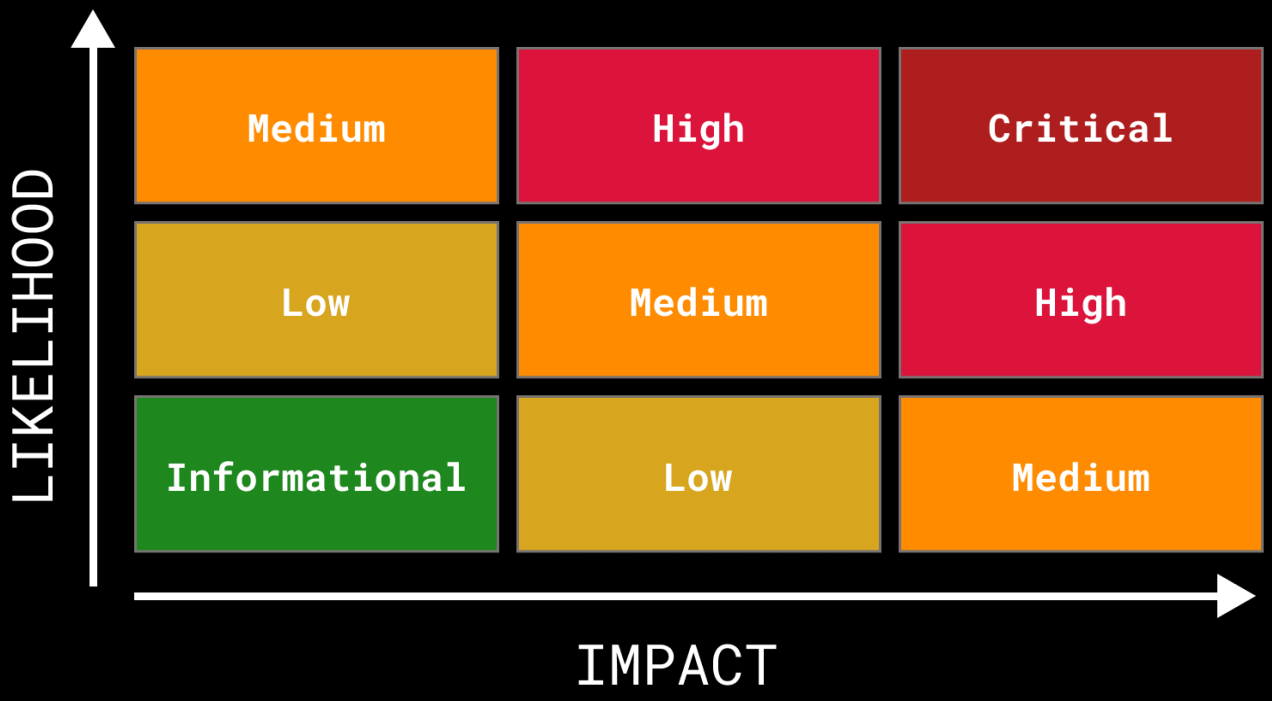
2. Acknowledged
   The issue has been reviewed and accepted by the team but remains unresolved.

3. Partially Resolved
   The issue has been mitigated to some extent, but residual risks or concerns persist.

4. Resolved
   The issue has been fully addressed, with no further action required.

5. Closed
   The issue is no longer relevant or actionable, regardless of resolution.

Risk Matrix


LIKELIHOOD
Medium
High
Critical
Low
Medium
High
Informational
Low
Medium
IMPACT

Summary of Findings

CODE	ISSUE	SEVERITY	STATUS
FKGG001	Inadequate Ownership Validation and Over-Centralized Withdrawals	Medium	Acknowledged
FKGG002	Insufficient Checks on Fee and Threshold Limits	Low	Acknowledged
FKGG003	Naming Ambiguity and Inconsistent Specification for Collateral	Low	Acknowledged
FKGG004	Redundant Checks in Deposit Function	Info	Acknowledged

Findings Overview

Severity: Medium	Impact: High	Likelihood: Low
FKGG001	**Inadequate Ownership Validation and Over-Centralized Withdrawals**	
File Name	PlatformCredits.sol	
Status	Acknowledged	

Description:

The withdrawal mechanism exhibits two intertwined issues:

1. Inadequate Token Ownership Validation:
   The withdraw() method validates the withdrawal by checking only the msg.sender's current balance of PC tokens (via the ERC20 balanceOf call). However, the contract maintains a separate persistent record of user deposits in the totalDeposits mapping. This separation means that the withdrawal process does not verify that the PC tokens being burned were originally minted through a user's own deposit. Consequently, PC tokens could be transferred among users despite the documentation's intent that they are non-transferable and solely used for trading on the platform. This lack of cross-verification undermines the integrity of the deposit–withdraw cycle.

2. Over-Centralized Withdrawal Approval:
   The contract enforces that only users approved by an operator can execute a withdrawal. This is done via the withdrawApproval mapping that is updated only through calls to approveWithdraw() or approveWithdrawWithSig(), which can only be invoked by addresses with the OPERATOR_ROLE. Such a centralized mechanism means that even if a user holds sufficient PC tokens, they cannot withdraw their underlying collateral unless explicitly approved by an operator. This pattern imposes a high degree of centralization, potentially allowing operator misuse or introducing delays and additional administrative overhead that conflict with the intended decentralized, self-service design.

   - Additionally, this adds a layer of operational complexity where each deposit requires subsequent withdrawal approval calls which will add in time and cost complexity for the protocol.

Impact on Protocol and Users:

1. Token Ownership Validation:

   Users might end up with PC tokens in their wallets that are not recognized as valid deposits for withdrawal, thereby violating the promise of non-transferability and internal-only use of Platform Credits. This could lead to confusion and unexpected behavior, where transferred tokens enable withdrawals that are not reflective of the actual deposit history.

2. Centralized Approval Mechanism:
   Requiring operator approval for withdrawals centralizes control over user funds, potentially delaying withdrawals or, in worst-case scenarios, enabling an operator to maliciously block or allow withdrawals contrary to user intent. This undermines user trust and could lead to disputes or loss of funds if the central authority is compromised or misbehaves.

Code Affected

```
// Withdrawal function snippet (ownership validation issue)
function withdraw(
    address token,
    uint256 amount,
    address recipient
) external override {
    // Only checking msg.sender's current PC token balance
    uint256 currentBalance = balanceOf(msg.sender);
    if (currentBalance < amount) {
        revert InsufficientBalanceForWithdraw(currentBalance, amount);
    }
    // ... (additional withdrawal logic)
}

// Centralized approval mechanism for withdrawals
function approveWithdraw(address account) external onlyRole(OPERATOR_ROLE) {
    withdrawApproval[account] = true;
    emit WithdrawApproved(account, msg.sender);
}

function approveWithdrawWithSig(
    ApprovalMsg memory approvalMsg,
    bytes calldata operatorSig
) external {
    bytes32 typehash = hashApprovalMsg(approvalMsg);
    uint256 executedTimestamp = executedMsgs[typehash];

    if (executedTimestamp != 0) {
        revert MsgAlreadyExecuted(typehash, executedTimestamp);
    }

    address signer = ECDSA.recover(typehash, operatorSig);
    _checkRole(OPERATOR_ROLE, signer);

    withdrawApproval[approvalMsg.account] = true;
    executedMsgs[typehash] = block.timestamp;

    emit WithdrawApproved(approvalMsg.account, signer);
}
```

Recommendation:

Ensure that withdrawals validate that the PC tokens being burned correspond to the user's own deposit history (e.g. by cross-checking with the totalDeposits mapping) and redesign the withdrawal approval process to reduce centralized control, allowing withdrawals based on inherent deposit records rather than solely on operator approval.

Developer Response:

"The medium-severity issue labeled "Inadequate Ownership Validation and Over-Centralized Withdrawals" reflects our system's intended behavior:

(1) The concern around users being able to withdraw PC based on their balance rather than the amount deposited is aligned with our logic — users can earn PC not only through collateral deposits but also via credit codes and other mechanisms. As such, we do not enforce a strict check against totalDeposits.

(2) The withdrawal function requiring operator confirmation is also by design, as part of our system's business rules and internal controls."

Auditor Response:

Acknowledged with no further refute.

Severity: Low	Impact: Medium	Likelihood: Low
FKGG002	**Insufficient Checks on Fee and Threshold Limits**	
File Name	PlatformCredits.sol	
Status	Acknowledged	

Description:

The contract lacks adequate validations for critical operational parameters during both initialization and subsequent updates. Specifically, the following issues were identified:

- Fee Misconfiguration:
  The contract inherits from the FeeManager by calling FeeManager(configuredFeeBps, configuredFeeReceiver) in the constructor without verifying that the provided fee basis points (configuredFeeBps) fall within acceptable minimum or maximum limits. Without such checks, it is possible to initialize the contract with fee values that are either excessively high or inappropriately low, potentially disrupting the economic model and harming user experience.

- Unrestricted Threshold Settings:
  The variables minimumWithdraw and depositLimit are set directly in the constructor using the provided values and can later be updated via the configureMinimumWithdraw() and configureDepositLimit() methods. No validation is implemented to ensure that these values adhere to predefined business rules or safe operational limits as might be specified in the protocol's public documentation. This omission could lead to scenarios where the minimum withdrawal amount is set too low (or too high) or deposit limits do not align with the intended usage guidelines, causing discrepancies between the smart contract behavior and the documented standards.

Impact on Protocol and Users:

- Fee Misconfiguration:
  Setting the fee value without proper bounds can lead to economic imbalances. Excessively high fees may deter platform usage, while extremely low fees might result in insufficient revenue generation for the platform.

- Inconsistent Thresholds:
  Without validation, minimumWithdraw and depositLimit could be configured to values that do not match the intended operational or business requirements. This misalignment can result in withdrawal or deposit behaviors that contradict user expectations as outlined in the public documentation, potentially causing user dissatisfaction or operational risks.

Code Affected:

FeeManager Initialization:
In the constructor:

```
FeeManager(configuredFeeBps, configuredFeeReceiver)

constructor(uint256 configuredFeeBps, address configuredFeeReceiver) {
        if (configuredFeeReceiver == address(0)) revert NullFeeReceiver();

        feeBps = configuredFeeBps;
        feeReceiver = configuredFeeReceiver;
    }
```

Threshold Variables Initialization:
In the constructor:

```
minimumWithdraw = minWithdraw;
depositLimit = configuredDepositLimit;

Configuration Methods:


function configureMinimumWithdraw(uint256 updatedVal) public onlyRole(OPERATOR_ROLE)
{
    uint256 currentMinimumWithdraw = minimumWithdraw;
    minimumWithdraw = updatedVal;
    emit MinimumWithdrawUpdated(currentMinimumWithdraw, updatedVal);
}

function configureDepositLimit(uint256 newLimit) external onlyRole(OPERATOR_ROLE) {
    uint256 currentLimit = depositLimit;
    depositLimit = newLimit;
    emit DepositLimitUpdated(currentLimit, newLimit);
}
```

Recommendation:

Implement validation logic in both the constructor and the configuration methods to ensure that fee values, minimum withdrawal amounts, and deposit limits adhere to defined acceptable ranges. Additionally, update the public documentation to clearly specify these thresholds and their intended limits to ensure transparency and consistency with business requirements.

Developer Response:

“While they [low-severity findings] may introduce some degree of centralization, they are not security vulnerabilities and are manageable within the context of our operational model.”

Auditor Response:

Acknowledged with no further refute.

Severity: Low	Impact: Low	Likelihood: Low
FKGG003	**Naming Ambiguity and Inconsistent Specification for Collateral**	
File Name	PlatformCredits.sol	
Status	Acknowledged	

Description:

The deposit and withdraw functions use an amount parameter ambiguously. In these methods, amount represents the number of Platform Credits (PC) tokens to be minted or burned rather than the underlying collateral token amount. This can mislead users who expect the parameter to indicate the collateral (USDC) amount, as stated in the documentation.

Additionally, the contract supports deposits and withdrawals of multiple tokens, not just USDC, by implementing conversion logic using different rates and a PRECISION constant. The documentation, however, only describes a 1:1 pairing with USDC. This inconsistency between the implementation and the documentation may result in user confusion regarding which tokens are acceptable and how conversions are handled, especially given varying token decimals.

Impact:

Users may misinterpret how much collateral is required or returned, potentially leading to incorrect deposits or withdrawals. This could also result in unexpected behavior when non-USDC tokens (with different decimals) are used.

Code Affected:

```
// Conversion calculation using PRECISION for multi-token support
uint256 private constant PRECISION = 10 ** 8;

function calculateNeededToken(
    uint256 amount,
    uint256 rate
) public pure returns (uint256 collateralAmount) {
    return (amount * rate) / PRECISION;
}

// Deposit function snippet
function deposit(address token, uint256 amount) external override {
    uint256 rate = supportedTokens[token];
    // ... (additional logic)
    uint256 neededAmount = calculateNeededToken(amount, rate);
    // ...
}

// Withdraw function snippet with recipient parameter
function withdraw(
```

```
    address token,
    uint256 amount,
    address recipient
) external override {
    uint256 rate = supportedTokens[token];
    // ... (additional logic)
}
```

Recommendation:

Clarify the parameter naming and update the documentation to accurately reflect that the contract supports multiple tokens with conversion logic, not just USDC with a fixed 1:1 ratio.

Developer Response:

“While they [low-severity findings] may introduce some degree of centralization, they are not security vulnerabilities and are manageable within the context of our operational model.”

Auditor Response:

Acknowledged with no further refute.

Severity: Info	
FKGG004	**Redundant Checks in Deposit Function**
File Name	PlatformCredits.sol
Status	Open

Description:

In the deposit() method, the contract performs explicit checks for the collateral token's allowance and balance before calling safeTransferFrom. Since safeTransferFrom (provided by the SafeERC20 library) already reverts if the allowance or balance is insufficient, these preliminary checks are redundant. This redundancy leads to increased code complexity and unnecessary gas usage without providing additional security.

Impact:

While this redundancy does not introduce a security vulnerability, it results in minor gas inefficiencies and unnecessary complexity in the code.

Code Affected:

```
uint256 collateralAllowance = tokenContract.allowance(msg.sender, address(this));
if (collateralAllowance < neededAmount) {
    revert InsufficientAllowance(collateralAllowance, neededAmount);
}

uint256 collateralBalance = tokenContract.balanceOf(msg.sender);
if (collateralBalance < neededAmount) {
    revert InsufficientCollateralForDeposit(collateralBalance, neededAmount);
}
```

Recommendation:

Remove the redundant allowance and balance checks and rely on the inherent reversion of safeTransferFrom for these validations.

Disclaimer

ImmuneBytes' security audit services are designed to help identify and mitigate potential security risks in smart contracts and related systems. However, it is essential to recognize that no security audit can guarantee absolute protection against all possible security threats. The audit findings and recommendations are based on the information provided during the assessment.

Furthermore, the security landscape is dynamic and ever-evolving, with new vulnerabilities and threats emerging regularly. As a result, it is strongly advised to conduct multiple audits and establish robust bug bounty programs as part of a comprehensive and continuous security strategy. A single audit alone cannot protect against all potential risks, making ongoing vigilance and proactive risk management essential.

To ensure maximum security, the project must implement the recommendations provided in the audit report and regularly monitor its smart contracts for vulnerabilities. It is imperative to adopt appropriate risk management strategies and remain proactive in addressing potential threats as they arise.

Please note that auditors cannot be held liable for any security breaches, losses, or damages that may occur after the audit has been completed despite using audit services. The responsibility for implementing the recommendations and maintaining the ongoing security of the systems rests solely with the project.