QuillAudits

AUDIT REPORT

August 2025

For

Paycio

Table of Content

Executive Summary

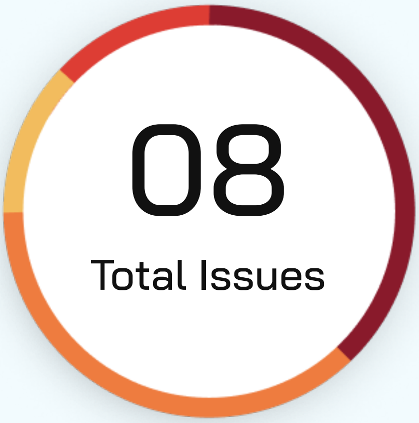
Project Name	Paycio
Protocol Type	ENS
Project URL	https://www.paycio.com/en/UCPI
Overview	AliasFundRouter is a Solidity smart contract that enables users to send cryptocurrency (ETH and ERC20 tokens) using human-readable aliases instead of wallet addresses. Users can register unique 3-32 character aliases that map to their wallet addresses, making transactions more user-friendly with memo support for transaction notes. The contract includes ENS (Ethereum Name Service) integration, allowing users to send funds to .eth domain names that resolve to wallet addresses. It features a multi-signature admin system requiring at least 2 signatures for administrative actions like adding/removing admins and blacklisting addresses for security.
Audit Scope	The scope of this Audit was to analyze the Paycio Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/paycioapp/UCPI_Contract/
Branch	Main
Contracts in Scope	ucpi_final_9_aug_ens.sol
Commit Hash	3da99d1730663216f6f0c23666a8ba65f862e280
Language	Solidity
Blockchain	EVM, TRON, and XRP.
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	18th August 2025 - 22nd August 2025
Updated Code Received	28th August 2025
Review 2	29th August 2025 - 30th August 2025

Fixed In	6086302aa3c0999b2bfaa1306268012c0c9719f4

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	3 (37.5%)
🟥	High	1 (12.5%)
🟧	Medium	3 (37.5%)
🟨	Low	1 (12.5%)
🟪	Informational	0(0.0%)

Severity

Issues

	🟥 Critical	🟥 High	🟧 Medium	🟨 Low	🟪 Informational
Open	0	0	0	0	0
Acknowledged	0	0	1	0	0
Partially Resolved	0	0	0	0	0
Resolved	3	1	2	1	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Administrators cannot execute blacklist actions due to timestamp mismatch vulnerability	Critical	Resolved
2	Administrators cannot blacklist malicious addresses due to missing signature collection mechanism	Critical	Resolved
3	Administrators can inadvertently grant administrative privileges to malicious addresses during blacklist operations	Critical	Resolved
4	Incorrect ENS Namehash Implementation	High	Resolved
5	Use a safe transfer helper library for ERC20 transfers	Medium	Resolved
6	Contract Can Permanently Lock Ether, Affecting Admin Recoverability	Medium	Resolved
7	Front-Running Vulnerability in Alias Registration	Medium	Acknowledged
8	Floating and Outdated Pragma	Low	Resolved

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls

- ☑ **Missing Zero Address Validation**
- ☑ **Private modifier**
- ☑ **Revert/require functions**
- ☑ **Multiple Sends**
- ☑ **Using suicide**
- ☑ **Using delegatecall**
- ☑ **Upgradeable safety**
- ☑ **Using throw**
- ☑ **Using inline assembly**
- ☑ **Style guide violation**
- ☑ **Unsafe type inference**
- ☑ **Implicit visibility level**

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

🟥 Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

🟥 High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

🟧 Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

🟨 Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

🟪 Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open Security vulnerabilities identified that must be resolved and are currently unresolved.	**Resolved** These are the issues identified in the initial audit and have been successfully fixed.
Acknowledged Vulnerabilities which have been acknowledged but are yet to be resolved.	**Partially Resolved** Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

		Impact		
		🟥 High	🟧 Medium	🟨 Low
Likelihood	🟥 High	Critical	High	Medium
	🟧 Medium	High	Medium	Low
	🟨 Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Critical Severity Issues

Administrators cannot execute blacklist actions due to timestamp mismatch vulnerability

Resolved

Path

ucpi_final_9_aug_ens.sol

Function Name

`executeBlacklistAction()`

Description

The AliasFundRouter contract contains a critical implementation flaw in the blacklist execution mechanism that renders the executeBlacklistAction() function completely non-functional. The vulnerability stems from a timestamp mismatch between the action proposal and execution phases that prevents legitimate blacklist operations from being carried out.

When an administrator proposes a blacklist action through proposeBlacklist(), the system generates a unique action identifier using keccak256(abi.encodePacked("blacklist", _wallet, block.timestamp)), where the current block timestamp is incorporated into the hash generation. This action identifier is stored in the pendingAdminActions mapping and serves as the key for subsequent operations. However, when attempting to execute the blacklist action via executeBlacklistAction(), the function attempts to reconstruct the expected action identifier using the same hashing mechanism but with the current block timestamp, which will inevitably be different from the timestamp used during proposal creation.

The execution function contains the flawed logic bytes32 expectedBlacklistId = keccak256(abi.encodePacked(“blacklist”, action.targetAdmin, block.timestamp)) and then compares this newly generated hash with the original action identifier. Sinceblock timestamps advance with each new block, this comparison will always fail, causing the blacklist execution to revert even when all other conditions are met.

Impact

This vulnerability completely disables the contract’s blacklist functionality, creating severe security implications for the protocol. Malicious actors who should be blacklisted cannot be restricted from using the contract, allowing them to continue registering aliases, sending funds, and potentially exploiting other vulnerabilities. The broken blacklist mechanism undermines the contract’s security model and administrative control capabilities.

Likelihood: High

Recommendation

The primary fix involves restructuring the action identification system to eliminate timestamp dependency in the execution phase. Instead of reconstructing action identifiers during execution, the system should store action metadata directly and use consistent identification mechanisms.

QuillAudits' Response

The issue has been fixed by removing executeBlacklistAction() and handling blacklisting through _executeAdminAction()

Administrators cannot blacklist malicious addresses due to missing signature collection mechanism

Resolved

Path

ucpi_final_9_aug_ens.sol

Function Name

`proposeBlacklist()`

Description

The AliasFundRouter contract contains a critical implementation gap in its blacklist execution mechanism that permanently prevents administrators from blacklisting malicious addresses. The vulnerability stems from the absence of a proper signature collection function for blacklist and whitelist operations, making it impossible to meet the minimum signature requirements for execution.

When an administrator proposes a blacklist action through proposeBlacklist(), the function creates an AdminAction struct and sets the initial signature count to one (action.signatures = 1). However, the contract provides no mechanism for other administrators to add their signatures to this blacklist proposal.

The executeBlacklistAction() function requires action.signatures >= MIN_SIGNATURES (where MIN_SIGNATURES = 2) to proceed with execution. Since no function exists to increment the signature count for blacklist actions beyond the initial proposer's signature, the action.signatures value remains permanently at one. This creates an impossible condition where the signature requirement check require(action.signatures >= MIN_SIGNATURES, "Not enough signatures") will always evaluate to require(1 >= 2), causing the function to revert with "Not enough signatures" regardless of how many administrators attempt to execute the action.

Impact

This vulnerability completely disables the contract's ability to enforce security policies against malicious actors, creating a permanent security vulnerability that undermines the entire platform's integrity. Malicious users who should be blacklisted can continue to operate within the system indefinitely, registering deceptive aliases, executing fraudulent transactions, and potentially exploiting other contract vulnerabilities without any administrative recourse.

Likelihood: High

Recommendation

The remediation must establish a proper workflow that allows multiple administrators to sign blacklist proposals while maintaining separation from administrator management functions.

QuillAudits' Response

The bug has been fixed by handling blacklisting through _executeAdminAction()

Administrators can inadvertently grant administrative privileges to malicious addresses during blacklist operations

Resolved

Path

ucpi_final_9_aug_ens.sol

Function Name

`signAdminAction()`

Description

The AliasFundRouter contract contains a catastrophic vulnerability in its action execution system that causes blacklist proposals to be incorrectly processed as administrator additions when administrators inadvertently use the wrong signing function. The vulnerability arises from a fundamental design flaw where the signAdminAction() function serves dual purposes but applies administrator management logic to all action types, including blacklist operations.

When an administrator proposes to blacklist a malicious address through proposeBlacklist(), the system creates an AdminAction struct with isAdding = true and stores the malicious address in the targetAdmin field. The proposal receives its first signature from the proposer and awaits additional signatures from other administrators. However, if any administrator mistakenly calls signAdminAction() instead of the intended blacklist-specific signing function, the system processes the blacklist proposal using administrator management logic.

The signAdminAction() function automatically triggers _executeAdminAction() when the minimum signature threshold is reached, which exclusively handles administrator additions and removals. Since blacklist proposals set isAdding = true, the execution logic interprets this as a request to add a new administrator. The function checks if the target address already exists in the admin list, and if not, adds the malicious address as a new administrator with admins.push(Admin({wallet: action.targetAdmin, isActive: true})). This grants full administrative privileges to the address that was intended to be blacklisted.

Impact

When administrators attempt to protect the system by blacklisting a malicious address, they inadvertently grant that address complete control over the contract, creating a devastating reversal of security intentions.

Likelihood: High

Recommendation

The remediation must establish clear boundaries between administrator management and blacklist operations while eliminating any possibility of accidental privilege escalation.

QuillAudits' Response

This bug has been fixed by differentiating various actions inside _executeAdminAction()

High Severity Issues

Incorrect ENS Namehash Implementation

Resolved

Path

ucpi_final_9_aug_ens.sol

Function

`getNamehash()`

Description

The contract attempts to resolve ENS names via resolveENS(), which internally relies on a helper function getNamehash(). However, the implementation of getNamehash() is not a full ENS namehash algorithm. Instead, it only performs a single keccak256 hash of the entire string and concatenates it with the zero node.

This approach does not comply with the ENS standard for hierarchical namehashing, which recursively hashes each label of the ENS name (right-to-left).

Impact

All ENS name resolutions will fail, causing any attempts to send funds to ENS names to revert. This breaks a major advertised feature of the contract and could lead to user frustration and potential fund loss if users rely on ENS functionality.

Likelihood: HIGH

Recommendation

Replace the custom simplified implementation with the official ENS library for namehashing

QuillAudits' Response

The issue has been fixed by using ENS library for namehashing

Medium Severity Issues

Use a safe transfer helper library for ERC20 transfers

Resolved

Path

ucpi_final_9_aug_ens.sol

Path

`sendERC20ToAlias()`

Description

The sendERC20ToAlias function assumes that every ERC–20 token it handles conforms strictly to the OpenZeppelin–style interface and returns a Boolean flag on transfer. In practice a significant subset of production tokens, either return void (no data) or revert on failure while returning no value. When such a token is passed to transfer(IERC20 token), the high–level Solidity call expects a bool but receives zero bytes, causing ABI decoding to revert.

Contract Can Permanently Lock Ether, Affecting Admin Recoverability

Resolved

Path

ucpi_final_9_aug_ens.sol

Path

`receive()`

Description

The Contract contains a receive() function that enables it to accept native Ether deposits. However, it lacks any mechanism to withdraw or recover these funds once received. This is a critical oversight in contract design, especially for contracts that are not meant to hold user or operational balances in the form of native tokens. There is no function exposed that allows the contract owner or any authorized entity to transfer Ether out of the contract, and there is no automated refund or forwarding logic in place.

As a result, any Ether sent to the contract, whether intentionally (e.g., by a user mistaking the contract for a payable endpoint) or unintentionally (e.g., via self-destructs or malicious transfers), becomes permanently locked

Front-Running Vulnerability in Alias Registration

Acknowledged

Path

ucpi_final_9_aug_ens.sol

Path

`registerAlias()`

Description

The registerAlias function is vulnerable to front-running attacks. When a user submits a transaction to register a desirable alias, an attacker monitoring the mempool can see this transaction and submit their own transaction with a higher gas price to register the same alias first. This is particularly problematic for valuable or meaningful aliases that users might want to claim.

Low Severity Issues

Floating and Outdated Pragma

Resolved

Path

ucpi_final_9_aug_ens.sol

Function Name

`pragma`

Description

Locking the pragma prevents the contract from being compiled with outdated Solidity versions that might contain security flaws.
In this case the pragma is left open as >= 0.8.0, meaning the code could be compiled with any later compiler release, which re–introduces the very risk the lock is meant to avoid.
Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.29 pragma version

Functional Tests

Some of the tests performed are mentioned below:

- ✓ Contract should deploy successfully with valid developerWallet and ensRegistry
- ✓ Constructor should set developerWallet and add it as the first admin
- ✓ Should revert if developerWallet is zero address
- ✓ Should revert if ensRegistry is zero address
- ✓ Registering a valid alias should succeed and emit AliasRegistered
- ✓ Should revert if alias is already taken
- ✓ Should revert if address already has an alias
- ✓ Should revert if alias length is <3 or >32 characters
- ✓ Should revert if alias is a valid .eth ENS name
- ✓ Should identify .eth names as ENS
- ✓ Should resolve a valid ENS name to an address
- ✓ Should emit ENSResolved on successful resolution
- ✓ Should revert if ENS name resolves to address(0)
- ✓ Should revert if string is not a valid .eth name

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats
AliasFundRouter	Constructor / Initialization	- Can give control to an attacker or break ENS resolution. - Allows spoofing recipient addresses
	registerAlias	- Popular aliases can be sniped. - No protection beyond basic mapping check. - Prevented in code, but any bypass could confuse users. - Cannot prevent Sybil attacks natively; off-chain monitoring needed.
	Fund Transfers (ETH / ERC20)	- ETH uses .call{value:} → potentially reentrant fallback - Not protected by nonReentrant modifier. - If attacker controls ENS record, can front-run and change address. - Can fail silently or block recipient fallback execution. - May return false silently. - May return false silently. - Missing use of SafeERC20.safeTransferFrom.

Contract	Function	Threats
	proposeAdminAction, signAdminAction, _executeAdminAction	Admins may race to add/remove with conflicting actions. Collisions can overwrite pending actions. If multiple admins collude or are compromised. Same actionId logic (based on timestamp) is weak for replays or future reuse.
	Blacklist / Whitelist System	- Overloading isAdding in both admin and blacklist contexts is ambiguous. - Can be abused if multiple admins collude. - Relies on block.timestamp which is manipulatable within blocks.

Closing Summary

In this report, we have considered the security of Paycio. We performed our audit according to the procedure described above.

Issues of Critical, High, Medium, and Low severity were found.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

August 2025

For

Paycio

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com