shieldify

Colb Finance

USC OffRamp

SECURITY REVIEW

Date: 16 October 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Colb Finance - USC OffRamp

Overview

The OffRamp contract serves as the core redemption vehicle for the Colb ecosystem, enabling users to convert $USC stablecoins back into fiat through a two-step off-ramp process. The system supports two request types (OffRamp, Payment) with chain-specific request IDs, allowing users to initiate different types of redemption operation. Through its integration with StableManagement for token burning and Whitelist contract, OffRamp maintains a regulated token management while ensuring that only verified users can participate in redemption operations.

Access Control

The OffRamp integrates the Colb Whitelist contract to ensure only verified accounts can create redemption requests, providing regulatory compliance and KYC/AML capabilities. The contract implements role-based permissions with administrative roles:
- Operators: Manage request completion and cancellation operations
- Administrators: Control contract configuration, treasury settings, and system parameters

The system enforces strict access controls on all administrative functions, ensuring that only authorized personnel can modify critical system parameters or complete redemption requests.

Treasury Management

The system supports configurable treasury addresses with authorization flags, allowing different redemption scenarios to use different treasury addresses. Treasury addresses can be set by administrators and marked as authorized for completion operations.

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 7 days, with a total of 112 hours dedicated to the audit by two researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying two Medium and seven Low severity issues, mainly related to an unnecessary cap inflation and unbounded cap growth, locked funds of old token by state management updates, missing validation checks in several functions and wrong gas limit configuration.

The Colb Finance team has done a great job with their test suite and provided support and responses to all of the questions that the Shieldify researchers had.

5.1 Protocol Summary

Project Name	**Colb Finance - USC OffRamp**
Repository	SmartContracts
Type of Project	**RWA, Pre-IPO, USC OffRamp**
Security Review Timeline	**7 days**
Review Commit Hash	e37c0a9c947658cab28f9432df75a4cc9cc60130
Fixes Review Commit Hash	b6d8bd87008c951db3fed81f4e948b8d28e73afe

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
contracts/ccip/PortalCCIP.sol	205
contracts/ramp/OffRamp.sol	234
interfaces/ccip/IPortalCCIP.sol	34
interfaces/ramp/IOffRamp.sol	41
Total	**514**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Medium** issues: **2**
- **Low** issues: **7**
- **Info** issues: **2**

ID	Title	Severity	Status
[M-01]	Unnecessary Cap Inflation and Unbounded Cap Growth	Medium	Fixed
[M-02]	Funds of Old Token Are Locked When `StableManagement` Is Updated	Medium	Fixed
[L-01]	Missing Zero Address Validation Causes Permanent Token Loss	Low	Fixed
[L-02]	Inadequate Gas Limit Configuration and Validation	Low	Acknowledged
[L-03]	Missing Escape Hatch for Failed CCIP Messages	Low	Acknowledged
[L-04]	Unused Timestamp Field in `sendDirect()`	Low	Fixed
[L-05]	Cross-Chain Blacklist Bypass Leads to Locked User Funds	Low	Acknowledged
[L-06]	Unsafe ERC20 Approval Usage	Low	Fixed
[L-07]	No Validation on Gas Limit Value	Low	Fixed
[I-01]	Unused Treasury Variable	Info	Fixed
[I-02]	RequestId Race Condition Due to `feeRate` Inclusion in Hash	Info	Fixed

7. Findings

[M-01] Unnecessary Cap Inflation and Unbounded Cap Growth

Severity

Medium Risk

Description

The `_setNewCap()` function in `PortalCCIP` unconditionally increases the token supply cap on every incoming cross-chain message, even when sufficient headroom already exists. This leads to unbounded cap growth that defeats the security purpose of having a cap mechanism, creates an indirect method to bypass cap restrictions, and wastes gas on unnecessary state changes.

Location of Affected Code

File: contracts/ccip/PortalCCIP.sol#L439

```
function _setNewCap(uint256 amount) internal {
    uint256 cap = stableManagement.cap();           // Current cap
    uint256 totalSupply = usc.totalSupply();        // Current total
       supply
    uint256 top = cap > totalSupply ? cap : totalSupply;  // Take maximum

    //   ALWAYS increases cap, even when unnecessary
    stableManagement.setNewCap(top + amount);
}
```

Impact

Can security control gets defeated

Recommendation

Only increase the cap when actually needed.

Team Response

Fixed.

[M-02] Funds of Old Token Are Locked When `StableManagement` is Updated

Severity

Medium Risk

Description

The `setStableManagement()` function in the `OffRamp` contract allows an authorized role to update the `stableManagement` variable, which also updates the `usc` token used for all off-ramp operations.

This design introduces a critical consistency issue for pending requests that were created before the update. Each off-ramp request implicitly assumes that the contract's `usc` reference continues to point to the same ERC20 token that was originally transferred by the user.

If `setStableManagement()` is called while requests are still pending: - The contract now references a new token address (`usc` from the new `stableManagement`). - Pending requests still correspond to the old token, but the contract's functions (`completeRequest()` and `cancelRequest()`) will interact with the **new token**. - As a result, these pending requests cannot be completed or cancelled, and the contract will attempt to transfer or burn the wrong token, leading to reverts or permanently locked funds.

Example scenario

1. A user creates a request when `usc` = TokenA. The contract holds 100 TokenA.
2. An operator later updates `stableManagement`, and `usc` now points to TokenB.
3. When trying to complete or cancel the pending request, the contract calls `usc.safeTransfer(...)` using TokenB, but the contract only holds TokenA.
4. The operation fails, leaving the request permanently stuck and user funds locked.

Location of Affected Code

File: contracts/ramp/OffRamp.sol#L177

```
function setStableManagement(IStableManagement newStableManagement)
    external {
     RoleChecker.hasRole(roleManager, OFF_RAMP_CONTROLLER_ROLE);

     if (address(newStableManagement) == address(0)) {
         revert ZeroAddress();
     }

     address uscAddress = address(newStableManagement.token());

     if (uscAddress == address(0)) {
         revert ZeroAddress();
     }

     emit StableManagementUpdated(address(stableManagement), address(
        newStableManagement), uscAddress);

     stableManagement = newStableManagement;
     usc = IERC20(uscAddress);
}
```

Impact

- **Permanent Fund Loss**: All pending requests at the time of the `stableManagement` update become permanently stuck. Users cannot recover their locked funds through `completeRequest()` or `cancelRequest()`, as both functions will revert when attempting to transfer the wrong token.

- **Protocol Disruption**: Every pending request is affected simultaneously. If multiple users have active requests during the update, all of them will lose access to their funds, potentially affecting a large number of users and significant capital.

Recommendations

Prevent updates while requests are pending: - Introduce a `pendingRequestsCounter` that increments on `createRequest()` and decrements on `completeRequest()` / `cancelRequest()` . - Require this counter to be zero before allowing `setStableManagement()` to execute. `solidity require(pendingRequestsCounter == 0, "Pending requests exist");`

Team Response

Fixed.

[L-01] Missing Zero Address Validation Causes Permanent Token Loss

Severity

Low Risk

Description

The `PortalCCIP` contract contains a critical vulnerability where tokens can be permanently lost during cross-chain bridging operations. The issue arises from the missing validation of the `destinationAddress` parameter before burning tokens in the `sendDirect()` function.

When a user initiates a bridge transaction via `sendDirect()`, the contract:

- Does NOT validate if `destinationAddress` is `address(0)`
- Burns the tokens on the source chain
- Sends a CCIP message to the destination chain

However, when the CCIP message arrives at the destination chain, `ccipReceive()` performs validation:

```
if (destinationAddress == address(0)) {
    revert InvalidDestinationAddress();
}
```

This causes the transaction to revert on the destination chain AFTER tokens have already been burned on the source chain, resulting in permanent loss of funds with no recovery mechanism.

Location of Affected Code

File: contracts/ccip/PortalCCIP.sol#L342-L345

```
function ccipReceive(Client.Any2EVMMessage calldata message) external
    whenNotPaused {
  // code
  // validate destination address
  if (destinationAddress == address(0)) {
      revert InvalidDestinationAddress();
  }
  // code
}
```

Impact

Users lose their bridged tokens with no recourse.

Recommendation

Add `destinationAddress` validation before burning.

Team Response

Fixed.

[L-02] Inadequate Gas Limit Configuration and Validation

Severity

Low Risk

Description

The `PortalCCIP` contract implements an insufficient gas limit configuration that violates CCIP best practices and exposes users to fund loss and excessive fees. The gas limit can be set to invalid values (including 0 or extremely high numbers) without bounds checking, lacks per-chain customization, and has no fallback mechanism for estimation failures.

In `initialize()` function, there is a Hardcoded Initial Value for `gaslimit`, which may be too low or too high due as the transaction might fail or users need to give more gas, which is not refunded.

Location of Affected Code

File: contracts/ccip/PortalCCIP.sol#L159

```
function initialize(
    IStableManagement _stableManagement,
    address _roleManager,
    address _treasury,
    uint64 _ccipChainId,
    address _ccipRouter
) public initializer {
    // code
    gasLimit = 300_000;
}
```

Impact

Message Execution Failures or Excessive Fee Overpayment

Recommendation

Create Dynamic Gas Estimation with Fallback.

Team Response

Acknowledged.

[L-03] Missing Escape Hatch for Failed CCIP Messages

Severity

Low Risk

Description

The `PortalCCIP` contract lacks any mechanism to recover from failed `ccipReceive()` executions, violating CCIP best practices for "Decoupling CCIP Message Reception and Business Logic." When a CCIP message delivery fails for any reason, user funds are permanently and irreversibly lost because tokens are burned on the source chain but never minted on the destination chain. There is no retry mechanism, no manual execution option, and no admin override to recover stuck messages.

Location of Affected Code

File: contracts/ccip/PortalCCIP.sol#L317

```
function ccipReceive(Client.Any2EVMMessage calldata message) external
    whenNotPaused {
     if (msg.sender != address(ccipRouter)) revert InvalidRouter(msg.
        sender);

     address srcAddress = abi.decode(message.sender, (address));

     if (srcAddress != trustedRemotes[message.sourceChainSelector]) {
         revert Unauthorized();
     }

     if (processedMessages[message.messageId]) {
         revert MessageAlreadyProcessed();
     }

     processedMessages[message.messageId] = true;

     (
         address sender,
         address sourceToken,
         uint256 amount,
         address destinationAddress
     ) = abi.decode(message.data, (address, address, uint256, address));

     if (destinationAddress == address(0)) {
         revert InvalidDestinationAddress();
     }

     if (blacklisted[sender] || blacklisted[destinationAddress]) {
         revert AddressBlacklisted();
     }

     _setNewCap(amount);

     stableManagement.mintToken(destinationAddress, amount);

     emit PortalRequestCompleted(message.messageId, sourceToken, sender,
        amount, destinationAddress);
}
```

Impact

Loss of funds as there is no recovery mechanism unless from the explorer.

Recommendation

Loss of funds may occur since there is no recovery mechanism, except through manual intervention via an explorer.

Team Response

Acknowledged.

[L-04] Unused Timestamp Field in `sendDirect()`

Severity

Low Risk

Description

The `CCIPMessage` struct stores a `timestamp` field that records when each cross-chain message is sent, but this timestamp is never used anywhere in the contract. This creates multiple issues: inability to detect and handle stale messages, no time-based validation, missing operational insights, wasted storage costs, and a lack of security controls that could prevent certain attack vectors. The `timestamp` is stored but provides no value, representing both wasted gas and missed security opportunities.

Location of Affected Code

File: contracts/ccip/PortalCCIP.sol#L257

```
struct CCIPMessage {
    address destinationAddress;
    uint64 destinationChainSelector;
    address sender;
    uint256 amount;
    uint256 timestamp;  //  Stored but never used
}
```

Impact

Paying for gas to store data that's never used.

Recommendation

Remove the `timestamp` if it's of no use.

Team Response

Fixed.

[L-05] Cross-Chain Blacklist Bypass Leads to Locked User Funds

Severity

Low Risk

Description

In `PortalCCIP`, users can initiate a cross-chain transfer via `sendDirect()`, which burns their USC tokens immediately. However, if the user or destination address gets blacklisted after the burn but before the `ccipReceive()` execution on the destination chain, the tokens will never be minted. The burned tokens are permanently lost with no recovery mechanism, effectively locking user funds forever. This creates an asymmetric risk where tokens are burned immediately, but minting can be blocked indefinitely.

Location of Affected Code

File: contracts/ccip/PortalCCIP.sol#L279

```
function sendDirect(uint256 amount, uint64 destinationChainSelector,
    address destinationAddress) external payable whenNotPaused
    nonReentrant {
  // code
  usc.safeTransferFrom(sender, address(this), amount);
  usc.approve(address(stableManagement), amount);
  stableManagement.burnTokenFrom(address(this), amount); // Tokens burned
      immediately
  // code
}
```

File: contracts/ccip/PortalCCIP.sol#L349

```
function ccipReceive(Client.Any2EVMMessage calldata message) external
    whenNotPaused {
  // code
  // Later in ccipReceive on destination chain:
  if (blacklisted[sender] || blacklisted[destinationAddress]) {
      revert AddressBlacklisted(); // Minting blocked, tokens already
          burned
  }
  // code
}
```

Impact

Legitimate users could permanently lose their USC tokens if blacklisted between initiating and receiving a cross-chain transfer.

Recommendation

Add a recovery system such that user funds do not get stuck.

Team Response

Acknowledged.

[L-06] Unsafe ERC20 Approval Usage

Severity

Low Risk

Description

The contract directly calls `usc.approve(address(stableManagement), amount))` without using `SafeERC20.safeApprove()`. While `IERC20.approve()` is a standard ERC-20 function, several tokens do not strictly follow the ERC-20 spec, they return no boolean value or revert when allowance is non-zero, leading to potential transaction reverts or unexpected approval failures.

Using OpenZeppelin's `SafeERC20.safeApprove()` ensures compatibility across all token implementations and safely handles non-standard return behaviors.

Location of Affected Code

File: contracts/ccip/PortalCCIP.sol#L281

```
function sendDirect(uint256 amount, uint64 destinationChainSelector,
    address destinationAddress) external payable whenNotPaused
    nonReentrant {
  // code
  usc.approve(address(stableManagement), amount);
  // code
}
```

Impact

The contract may revert unexpectedly when interacting with non-standard ERC-20 tokens

Recommendation

Replace the direct call with the safe wrapper provided by `SafeERC20`

Team Response

Fixed.

[L-07] No Validation on Gas Limit Value

Severity

Low Risk

Description

The `setGasLimit()` function in `PortalCCIP` does not validate that the new gas limit is within reasonable bounds. An operator could accidentally set the gas limit to 0 or an excessively high value, which could cause CCIP messages to fail or incur unexpectedly high fees.

Location of Affected Code

File: contracts/ccip/PortalCCIP.sol#L242

```
function setGasLimit(uint256 newGasLimit) external {
    RoleChecker.hasRole(roleManager, PORTAL_OPERATOR_ROLE);

    emit GasLimitUpdated(gasLimit, newGasLimit);

    gasLimit = newGasLimit;
}
```

Impact

- A gas limit of 0 would cause all CCIP messages to fail on the destination chain
- An excessively high gas limit could result in unnecessarily high fees
- Users might lose funds if messages fail due to misconfigured gas limits

Recommendation

Add validation to ensure the gas limit is within reasonable bounds:

```
function setGasLimit(uint256 newGasLimit) external {
    RoleChecker.hasRole(roleManager, PORTAL_OPERATOR_ROLE);

    if (newGasLimit == 0 || newGasLimit > 5_000_000) {
        revert InvalidGasLimit();
    }

    emit GasLimitUpdated(gasLimit, newGasLimit);

    gasLimit = newGasLimit;
}
```

Team Response

Fixed.

[I-01] Unused Treasury Variable

Severity

Informational Risk

Description

The `treasury` state variable in `PortalCCIP` is set during initialization but is never used anywhere in the contract. This represents dead code and could indicate incomplete implementation or a design change that wasn't fully cleaned up.

Location of Affected Code

File: contracts/ccip/PortalCCIP.sol#L79

```
uint256 public gasLimit; // Gas limit for CCIP messages
address public treasury; // Treasury to receive fees  // <-- declared but
    unused
IRouterClient public ccipRouter; // CCIP router address per chain

// In initialize:
treasury = _treasury;
```

Impact

- Wastes storage slot (gas inefficiency)
- Creates confusion about the contract's intended functionality
- May indicate an incomplete fee collection mechanism

Recommendation

Either implement the treasury functionality (if fees should be collected) or remove the variable entirely:

```
// Remove from state variables
// Remove from initialize parameters
// Remove from initialize function
```

If the treasury is intended for future fee collection, add appropriate documentation explaining the planned usage.

Team Response

Fixed.

[I-02] RequestId Race Condition Due to `feeRate` Inclusion in Hash

Severity

Informational Risk

Description

The `getRequestId()` function includes the `feeRate` parameter in the requestId hash calculation. This creates a race condition where the requestId cannot be reliably predicted if an admin changes a user's custom fee rate between transaction submission and execution.

The issue occurs because: 1. A user prepares a transaction using the current effective fee rate 2. Admin changes the user's custom fee rate (via `setAccountFeeRate()`) before the transaction is mined 3. The transaction executes with the new fee rate, generating a different requestId than expected 4. The generated requestId differs from what was calculated before submission

The `feeRate` parameter is redundant in the requestId calculation because the `nonce` already guarantees uniqueness per user. Including the fee rate adds unnecessary complexity and unpredictability to the ID generation process without providing any additional security or uniqueness guarantees.

Example Scenario: 1. User prepares transaction: `createRequest(100 tokens, OffRamp)` with current feeRate = 50 bps 2. Expected `requestId = hash(..., feeRate=50)` is calculated 3. Admin changes the user's custom fee to 100 bps before the transaction mines 4. Transaction executes with `feeRate = 100` bps 5. Actual `requestId = hash(..., feeRate=100)` ≠ expected requestId 6. The created request cannot be located using the pre-calculated ID

Location of Affected Code

File: contracts/ramp/OffRamp.sol#L438

```
function getRequestId(address sender, uint256 amount, uint256 nonce, Type
    requestType, uint256 feeRate) public view returns (bytes32) {
    return keccak256(abi.encodePacked(address(this), sender, amount, usc,
        nonce, requestType, feeRate, block.chainid));
}
```

Impact

Request Tracking Failures: Applications that pre-calculate request IDs will fail to locate requests when fee rates change between calculation and execution, resulting in confusion and operational difficulties.

Recommendation

Remove the `feeRate` parameter from the requestId calculation, as the `nonce` already guarantees uniqueness per user. The fee information is still stored in `request.fee` and accessible when needed.

Team Response

Fixed.

shieldify

Thank you!