

CHAINLINK

Chainlink CCIP Maple

Security Assessment Report

Version: 2.6

January, 2026

Contents

Introduction **2**
- Disclaimer . . . 2
- Document Structure . . . 2
- Overview . . . 2

Security Assessment Summary **3**
- Scope . . . 3
- Approach . . . 3
- Coverage Limitations . . . 3
- Findings Summary . . . 4

Detailed Findings **5**

Summary of Findings **6**
- Stale Share Amount In `removeRedemptionRequests()` After Partial Processing . . . 7
- Insufficient ETH For CCIP Fees Causes Stuck Deposits Instead Of Early Reverts . . . 8
- Griefing Risk Between `retryFailedMessage()` And `recoverFailedTokensBackToSender()` . . . 10
- Redemption Cancellation Does Not Refund Protocol Fee . . . 12
- Redemption Fees Not Stable In Asset Terms . . . 14
- Failed Messages With Malformed Payloads Cannot Be Recovered . . . 16
- Insufficient Deposit Fee Condition May Allow Zero Deposits . . . 18
- Missing Slippage Protection On Cross Chain Deposits And Redemptions . . . 19
- Insufficient Validation Of Fee Configuration Relative To CCIP Costs . . . 21
- Insufficient Balance Check In `withdrawNative()` . . . 23
- Ambiguous Fee Semantics In `getCollectedFees()` . . . 24
- Failed Messages Getter `getFailedMessages()` Returns Resolved Entries . . . 25
- Interface Inconsistency In `removeSharesById()` Return Values . . . 27

A Vulnerability Severity Classification **28**

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Chainlink components in scope. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Chainlink components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Chainlink components in scope.

Overview

CCIP Universal Cross-Chain Bridge is a Chainlink CCIP-powered bridge enabling direct cross-chain deposits into Maple Finance pools with automatic share token returns, redemptions and comprehensive recovery mechanisms.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the maple-labs/maple-cross-chain-receiver repository.

The scope of this time-boxed review was strictly limited to files at commit c125a61.

Retesting was performed on commit 680c301.

`CCIPUniversalUpgradeable.sol` was renamed to `MapleCCIPReceiver.sol` in the retesting commit.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

For the Solidity components, the manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 13 issues during this assessment. Categorised by their severity:

- Low: 6 issues.
- Informational: 7 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Chainlink components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
CPU-01	Stale Share Amount In `removeRedemptionRequests()` After Partial Processing	Low	Closed
CPU-02	Insufficient ETH For CCIP Fees Causes Stuck Deposits Instead Of Early Reverts	Low	Resolved
CPU-03	Griefing Risk Between `retryFailedMessage()` And `recoverFailedTokensBackToSender()`	Low	Resolved
CPU-04	Redemption Cancellation Does Not Refund Protocol Fee	Low	Closed
CPU-05	Redemption Fees Not Stable In Asset Terms	Low	Closed
CPU-06	Failed Messages With Malformed Payloads Cannot Be Recovered	Low	Resolved
CPU-07	Insufficient Deposit Fee Condition May Allow Zero Deposits	Informational	Closed
CPU-08	Missing Slippage Protection On Cross Chain Deposits And Redemptions	Informational	Closed
CPU-09	Insufficient Validation Of Fee Configuration Relative To CCIP Costs	Informational	Closed
CPU-10	Insufficient Balance Check In `withdrawNative()`	Informational	Closed
CPU-11	Ambiguous Fee Semantics In `getCollectedFees()`	Informational	Resolved
CPU-12	Failed Messages Getter `getFailedMessages()` Returns Resolved Entries	Informational	Resolved
CPU-13	Interface Inconsistency In `removeSharesById()` Return Values	Informational	Resolved

CPU-01	Stale Share Amount In `removeRedemptionRequests()` After Partial Processing		
Asset	`CCIPUniversalUpgradeable.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `removeRedemptionRequests()` function uses a stale `request.shares` value when cancelling redemptions, which can revert if Maple has partially processed the request and reduced the remaining shares. When a redemption is requested, CCIP records the full share amount:

```
$.requestedRedemptions[lastRequestId] = RequestedRedemption({
    universalSenderAddress: _universalSenderAddress,
    shares: sharesToRedeem,
    pool: _pool,
    chainSelector: _chainSelector,
    isRedeemed: false
});
```

If Maple processes only part of the request (for example when `maximumSharesToProcess_` is less than the full request), it updates the remaining shares in its queue but CCIP's `requestedRedemptions[requestId].shares` still holds the original full amount. Later, when `removeRedemptionRequests()` tries to cancel, it passes this stale full amount to Maple:

```
// Call removeSharesById to remove all remaining shares
uint256 sharesReturned = IWithdrawalManagerLike(withdrawalManager).removeSharesById(
    requestId,
    request.shares
);
```

Maple's `removeSharesById()` validates that `sharesToRemove_ <= request_.shares`, where `request_.shares` is the current remaining amount (which may be less than the original). Since CCIP passes the original full amount, this check fails and the cancellation reverts, leaving users unable to cancel partially processed redemptions.

Recommendations

Query Maple for the current remaining shares for each request before calling `removeSharesById()`, and use that value instead of the stale `request.shares` stored in CCIP, so that cancellation works correctly even after partial processing has occurred.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Redemption operator aware of limitation, partial redemptions that later get cancelled are unlikely. Operator will handle edge cases appropriately when using `removeSharesById()`"*

CPU-02	Insufficient ETH For CCIP Fees Causes Stuck Deposits Instead Of Early Reverts		
Asset	`CCIPUniversalUpgradeable.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

When this contract lacks enough ETH to pay CCIP fees, the failure is detected only inside `_svmCcipSend()` or `_evmCcipSend()` after message processing has already succeeded, which causes `ccipReceive()` to mark the CCIP message as failed and leaves the user deposit stuck pending recovery instead of reverting the whole operation up front. The CCIP send helpers compute the fee and revert with `NotEnoughBalance()` if the contract balance is too low:

```
function _svmCcipSend(
    uint64 _destinationChainSelector,
    bytes32 _beneficiary,
    address _token,
    uint256 _amount
) internal returns (bytes32 messageId) {
    if (_amount == 0) revert AmountIsZero();
    // ...
    // Get the fee required to send the CCIP message
    uint256 fees = IRouterClient(CCIP_ROUTER).getFee(_destinationChainSelector, evm2AnyMessage);

    // Check ETH balance for fees
    if (fees > address(this).balance) revert NotEnoughBalance(address(this).balance, fees);

    // Approve the Router to spend tokens
    IERC20(_token).approve(CCIP_ROUTER, _amount);

    // Pay fees with ether
    messageId = IRouterClient(CCIP_ROUTER).ccipSend{value: fees}(_destinationChainSelector, evm2AnyMessage);
    // ...
}
```

```
function _evmCcipSend(
    uint64 _destinationChainSelector,
    address _beneficiary,
    address _token,
    uint256 _amount
) internal returns (bytes32 messageId) {
    if (_amount == 0) revert AmountIsZero();
    // ...
    // Get the fee required to send the CCIP message
    uint256 fees = IRouterClient(CCIP_ROUTER).getFee(_destinationChainSelector, evm2AnyMessage);

    // Check ETH balance for fees
    if (fees > address(this).balance) revert NotEnoughBalance(address(this).balance, fees);

    // Approve the Router to spend tokens
    IERC20(_token).approve(CCIP_ROUTER, _amount);

    // Pay fees with ether
    messageId = IRouterClient(CCIP_ROUTER).ccipSend{value: fees}(_destinationChainSelector, evm2AnyMessage);
    // ...
}
```

These reverts are caught by the outer `try/catch` in `ccipReceive()`, which marks the message as failed and stores it for recovery:

```
function ccipReceive(Client.Any2EVMMessage calldata message) external override onlyCCIPRouter nonReentrant {
    uint256 gasLimit = gasleft();
    unchecked {
        if (gasLimit < MIN_FAILED_MESSAGE_GAS) revert CCIPDefensiveReceiverInsufficientGas();
        gasLimit -= MIN_FAILED_MESSAGE_GAS;
    }

    try this.processMessage{gas: gasLimit}(message) {
        emit MessageSucceeded(message.messageId);
    } catch {
        CCIPUniversalStorage storage $ = _getCCIPUniversalStorage();

        // Store full message content
        $.s_messageContents[message.messageId] = message;
        $.s_failedMessages.set(message.messageId, uint256(ErrorCode.FAILED));

        emit MessageFailed(message.messageId, message);
    }
}
```

Because deposit and redemption processing can already have taken place before the CCIP fee check fails, users end up with state changes on Maple but no outbound CCIP transfer, and must rely on the recovery path instead of having the original operation revert early when there is insufficient ETH in this contract.

Recommendations

Update the flow so that CCIP fees are checked before doing any deposit or redemption work, and if this contract does not have enough ETH for the CCIP fee then reject the message immediately instead of processing it and relying on the failed-message recovery path.

Resolution

Added upfront ETH balance check in `_processDeposit()` before any state changes

New `_estimateCcipFeeForDeposit()` function estimates fees upfront, preventing partial state changes.

Fixed in commit 680c301

CPU-03	Griefing Risk Between `retryFailedMessage()` And `recoverFailedTokensBackToSender()`		
Asset	`CCIPUniversalUpgradeable.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `retryFailedMessage()` flow can be griefed by `recoverFailedTokensBackToSender()`, allowing a third party to prevent a failed message from ever being retried to Maple by racing or pre-empting it with a recovery back to the sender. Once recovery is triggered, the message state is set to `RESOLVED`, so subsequent or concurrent `retryFailedMessage()` calls will fail their state checks and cannot progress.

```
function recoverFailedTokensBackToSender(bytes32 messageId) external payable nonReentrant {
    CCIPUniversalStorage storage $ = _getCCIPUniversalStorage();

    // Prepare recovery data with failure state validation
    RecoveryMessageData memory recoveryData = _prepareRecoveryMessage(messageId, true);

    // Check if caller sent enough ETH for fees
    if (msg.value < recoveryData.fees) {
        revert NotEnoughBalance(msg.value, recoveryData.fees);
    }

    // Mark the message as resolved
    $.s_failedMessages.set(messageId, uint256(ErrorCode.RESOLVED));

    // Approve the Router to spend all tokens from the failed message
    for (uint256 i = 0; i < recoveryData.failedMessage.destTokenAmounts.length; i++) {
        IERC20(recoveryData.failedMessage.destTokenAmounts[i].token).approve(
            CCIP_ROUTER,
            recoveryData.failedMessage.destTokenAmounts[i].amount
        );
    }

    // Send the message through the router with ETH payment
    bytes32 recoveryMessageId = IRouterClient(CCIP_ROUTER).ccipSend{value: recoveryData.fees}(
        recoveryData.failedMessage.sourceChainSelector,
        recoveryData.evm2AnyMessage
    );
    // ...
}
```

```
function retryFailedMessage(bytes32 messageId) external payable nonReentrant {
    // Verify message and mark as resolved
    Client.Any2EVMMessage memory message = _verifyMessageFailedAndMarkResolved(messageId);

    // Process the message
    _processMessageInternal(message);

    emit MessageRecovered(messageId);
}
```

Because `_verifyMessageFailedAndMarkResolved()` requires the message to still be in `FAILED` state, any call to `recoverFailedTokensBackToSender()` that sets it to `RESOLVED` first will permanently block retries for that message, allowing griefing of the retry path even though there is no direct financial gain.

Recommendations

Align the retry and recovery flows so that `recoverFailedTokensBackToSender()` cannot be invoked by arbitrary third parties, for example by restricting it to the original message sender or a governance role, or by introducing a delay or policy window during which only controlled retry is allowed, to reduce the risk that unrelated actors can front-run legitimate retries and permanently divert failed messages away from Maple.

Resolution

`recoverFailedTokensBackToSender()` now attempts retry first before recovery.

Retry is attempted before recovery, preventing griefing attacks

Fixed in commit 680c301

CPU-04	Redemption Cancellation Does Not Refund Protocol Fee		
Asset	`CCIPUniversalUpgradeable.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

When a redemption request is cancelled via `removeRedemptionRequests()`, the user's protocol redemption fee (charged in shares) is not refunded, and the fee model is independent of the actual CCIP costs for cancelling. The redemption fee is charged up front in `_requestRedeem()` by converting the configured asset-denominated fee into `feeShares`, reducing the user's shares and crediting the protocol's fee balance:

```
uint256 redemptionFee = $.s_redemptionFees[assetToken];

uint256 sharesToRedeem = _shares;
feeShares = 0;

if (redemptionFee > 0) {
    feeShares = IPoolLike(_pool).convertToExitShares(redemptionFee);

    if (feeShares >= _shares) {
        revert InsufficientRedemptionFee(redemptionFee, IPoolLike(_pool).convertToExitAssets(_shares));
    }

    sharesToRedeem = _shares - feeShares;
}

try IPoolLike(_pool).requestRedeem(sharesToRedeem, address(this)) returns (uint256 escrowShares_) {
    // ...
    if (feeShares > 0) {
        $.s_tokenFeesBalances[_pool] += feeShares;
        emit CollectedFee(_pool, feeShares);
    }
} catch {
    revert RedeemRequestFailed();
}
```

Later, `removeRedemptionRequests()` returns shares to the user and charges CCIP fees via `msg.value`, but it does not adjust `s_tokenFeesBalances` or otherwise refund the redemption fee that was previously collected:

```
function removeRedemptionRequests(uint128[] calldata requestIds) external payable nonReentrant {
    CCIPUniversalStorage storage $ = _getCCIPUniversalStorage();
    address pool = $.s_pool;
    // ...
    for (uint256 i = 0; i < requestIds.length; i++) {
        uint128 requestId = requestIds[i];
        RequestedRedemption storage request = $.requestedRedemptions[requestId];
        // ...
        uint256 sharesReturned = IWithdrawalManagerLike(withdrawalManager).removeSharesById(
            requestId,
            request.shares
        );
        // ...
        uint256 ccipFee = IRouterClient(CCIP_ROUTER).getFee(request.chainSelector, ccipMessage);
        // store RemovalData including ccipFee
    }
    // ...
    for (uint256 i = 0; i < validCount; i++) {
        RemovalData memory removal = removalsData[i];
        $.requestedRedemptions[removal.requestId].isRedeemed = true;
        _universalCcipSend(
            removal.chainSelector,
            removal.universalSenderAddress,
            pool,
            removal.sharesReturned
        );
    }
    // refund excess ETH to caller
}
```

As a result, a user whose redemption is cancelled regains only the reduced `sharesReturned` while the protocol keeps the earlier redemption fee in `feeShares`, and the configured redemption fee amount is a global lump sum that does not directly track the per-request or per-chain CCIP removal cost, which can result in overcharging or undercharging relative to actual cancellation costs.

Recommendations

Align the redemption fee and cancellation flows so that users do not permanently pay the protocol redemption fee when their redemption is cancelled, and ensure that any protocol-level redemption or cancellation fee is structured and documented in a way that reflects the actual CCIP costs and chain or token specific conditions rather than relying on a single global lumpsum configuration.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Nominal fee covers operational costs (gas, CCIP fees, protocol overhead). Fee structure is documented and intentional, users understand fee is non-refundable"*

CPU-05	Redemption Fees Not Stable In Asset Terms		
Asset	`CCIPUniversalUpgradeable.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Redemption fees are configured in asset units via `s_redemptionFees[assetToken]` but are later converted to shares at the time of the redemption request, meaning that if the vault suffers losses and the share price decreases before fee withdrawal, the actual assets realised from those fee shares will be less than the configured fee. The code calculates `feeShares` based on the current exit rate:

```
uint256 redemptionFee = $.s_redemptionFees[assetToken];

uint256 sharesToRedeem = _shares;
feeShares = 0;

if (redemptionFee > 0) {
    // Calculate how many shares are needed for the fee using convertToExitShares
    feeShares = IPoolLike(_pool).convertToExitShares(redemptionFee);

    if (feeShares >= _shares) {
        revert InsufficientRedemptionFee(redemptionFee, IPoolLike(_pool).convertToExitAssets(_shares));
    }

    sharesToRedeem = _shares - feeShares;
}
```

These `feeShares` are then tracked as collected fees against the pool address, but future losses in the pool will reduce the asset value these shares ultimately represent, so the protocol may receive less underlying than implied by `s_redemptionFees[assetToken]`:

```
RequestedRedemption memory request = RequestedRedemption({
    universalSenderAddress: _universalSenderAddress,
    shares: sharesToRedeem,
    pool: _pool,
    chainSelector: _chainSelector,
    isRedeemed: false
});
$.requestedRedemptions[lastRequestId] = request;

if (feeShares > 0) {
    $.s_tokenFeesBalances[_pool] += feeShares;
    emit CollectedFee(_pool, feeShares);
}
```

Recommendations

Align redemption fee handling so that fees configured in `s_redemptionFees[assetToken]` are realised in asset units at the time a redemption is successfully executed, for example by deriving and deducting the fee from the redeemed asset amount immediately before the CCIP send back to the user instead of relying on share-denominated fees that may later convert to a lower asset amount.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Nominal fees cover operational costs, not meant to maintain precise asset value. Fee variance due to pool performance is acceptable given nominal nature of fees."*

CPU-06	Failed Messages With Malformed Payloads Cannot Be Recovered		
Asset	`CCIPUniversalUpgradeable.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Messages that fail because `abi.decode()` cannot parse `message.data` are marked as failed but can never be recovered, which can leave user funds permanently stuck in the contract. The initial processing decodes the message data, and if this reverts the `catch` block stores the full `Client.Any2EVMMessage` including the malformed data:

```
function _processMessageInternal(Client.Any2EVMMessage memory any2EvmMessage) internal {
    CCIPUniversalStorage storage $ = _getCCIPUniversalStorage();
    // ...
    emit MessageReceived(
        any2EvmMessage.messageId,
        any2EvmMessage.sourceChainSelector,
        any2EvmMessage.sender,
        any2EvmMessage.data,
        inboundToken,
        any2EvmMessage.destTokenAmounts[0].amount
    );

    // Decode the universal message data using our defined struct
    UniversalMessage memory message = abi.decode(any2EvmMessage.data, (UniversalMessage));
    // ...
}
```

Later, during recovery, `_prepareRecoveryMessage()` calls `_getSenderAddressFromFailedMessage()`, which attempts to decode the same malformed payload again and will revert for these messages, preventing `recoverFailedTokensBackToSender()` and `estimateRecoveryFee()` from completing successfully:

```
function _prepareRecoveryMessage(
    bytes32 messageId,
    bool validateFailureState
) internal view returns (RecoveryMessageData memory recoveryData) {
    recoveryData = _validateAndExtractRecoveryData(messageId, validateFailureState);

    // Get sender address
    recoveryData.universalSenderAddress = _getSenderAddressFromFailedMessage(recoveryData.failedMessage);
    // ...
}

function _getSenderAddressFromFailedMessage(
    Client.Any2EVMMessage memory failedMessage
) internal view returns (bytes32 universalSenderAddress) {
    UniversalMessage memory message = abi.decode(failedMessage.data, (UniversalMessage));
    return message.universalSenderAddress;
}
```

Recommendations

Avoid re-decoding the bad payload when rebuilding the recovery message.

Cache the normalized sender (`_normalizeToBytes32(message.sender)`) alongside the failed message before the first decode and reuse that value during recovery.

Resolution

Store normalized sender address when message fails, use stored value during recovery.

Added `s_normalizedSenders` mapping to track sender without decoding malformed data.

Fixed in commit 680c301

CPU-07	Insufficient Deposit Fee Condition May Allow Zero Deposits
Asset	`CCIPUniversalUpgradeable.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The deposit fee check in the cross-chain deposit flow uses a strict `>` comparison, which permits `feeAmount` equal to `totalAmount` and results in a zero `depositAmount` being forwarded to Maple.

```
uint256 feeAmount = $.s_depositFees[depositToken];

if (feeAmount > 0) {
    if (feeAmount > totalAmount) {
        revert InsufficientDepositFee(feeAmount, totalAmount);
    }
    // Collect the fee and update the deposit amount
    depositAmount = totalAmount - feeAmount;
    $.s_tokenFeesBalances[depositToken] += feeAmount;
    emit CollectedFee(depositToken, feeAmount);
}
```

Recommendations

Tighten the validation condition around deposit fees so that scenarios where `feeAmount` is equal to `totalAmount` are rejected at this layer instead of relying on downstream Maple reverts.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Invalid deposits are handled by downstream Maple contract validation."*

CPU-08	Missing Slippage Protection On Cross Chain Deposits And Redemptions
Asset	`CCIPUniversalUpgradeable.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The cross chain deposit and redemption flows do not provide any slippage protection from the user's perspective. Deposits use `_vaultDeposit()` to call the Maple pool without any user specified minimum shares, and redemptions use `_executeRedeem()` to convert shares back to assets without any user specified minimum assets, even though execution can be delayed and prices can move between message creation and processing.

```
function _vaultDeposit(address _token, uint256 _amount, address _pool) internal returns (uint256 shares) {
    // ...
    // Perform the deposit and receive shares
    try IPoolLike(_pool).deposit(_amount, address(this)) returns (uint256 mintedShares) {
        shares = mintedShares;
    } catch {
        revert DepositFailed();
    }
    // Validate that shares were actually minted
    if (shares == 0) {
        revert InsufficientShares(1, 0);
    }
    // ...
}
```

```
function _executeRedeem(uint128 _requestId, CCIPUniversalStorage storage $) internal {
    RequestedRedemption storage request = $.requestedRedemptions[_requestId];
    if (request.pool == address(0) || request.isRedeemed) {
        return;
    }

    uint256 amount = IPoolLike(request.pool).redeem(request.shares, address(this), address(this));
    request.isRedeemed = true;
    uint64 destinationChainSelector = request.chainSelector;
    bytes32 beneficiary = request.universalSenderAddress;
    address token = address(IPoolLike(request.pool).asset());
    _universalCcipSend(
        destinationChainSelector,
        beneficiary,
        token,
        amount
    );
    // ...
}
```

In both directions, the protocol executes based on encoded instructions that do not carry any minimum acceptable outcome, so users may receive fewer shares or fewer assets than expected due to price or conversion ratio changes between the origin chain action and destination chain execution, with no way to reject unfavourable execution.

Recommendations

Provide slippage protection for both cross chain deposits and redemptions by extending the encoded message format to carry user defined constraints (for example minimum acceptable shares on deposit and minimum acceptable assets

on redemption) and enforcing those constraints when interacting with the Maple pool.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Cross-chain operations inherently involve delays, slippage protection would add significant complexity. Users should be aware that execution may occur at different prices than when message was created"*

CPU-09	Insufficient Validation Of Fee Configuration Relative To CCIP Costs
Asset	`CCIPUniversalUpgradeable.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Deposit and redemption fees are set via `setTokenDepositFee()` and `setTokenRedemptionFee()` with no built-in validation against the CCIP fees the contract pays from its own ETH balance when sending or retrying messages. Fee schedules should therefore be calibrated to cover CCIP send and retry costs (potentially with buffer) so the contract does not end up subsidizing users in adverse or high-retry scenarios.

```
function setTokenDepositFee(address token, uint256 feeAmount) external onlyRole(DEFAULT_ADMIN_ROLE) {
    CCIPUniversalStorage storage $ = _getCCIPUniversalStorage();
    $.s_depositFees[token] = feeAmount;
    emit TokenDepositFeeSet(token, feeAmount);
}

/// @notice Sets the fee amount to be collected for a specific token redemption
/// @param token The token address
/// @param feeAmount The fee amount to collect (in token units)
/// @dev Only callable by DEFAULT_ADMIN_ROLE
function setTokenRedemptionFee(address token, uint256 feeAmount) external onlyRole(DEFAULT_ADMIN_ROLE) {
    CCIPUniversalStorage storage $ = _getCCIPUniversalStorage();
    $.s_redemptionFees[token] = feeAmount;
    emit TokenRedemptionFeeSet(token, feeAmount);
}
```

The contract pays CCIP fees from its own ETH balance when sending messages, and retries failed messages using its own ETH-funded execution, as shown in `_svmCcipSend()` and `retryFailedMessage()`:

```
function _svmCcipSend(
    uint64 _destinationChainSelector,
    bytes32 _beneficiary,
    address _token,
    uint256 _amount
) internal returns (bytes32 messageId) {
    if (_amount == 0) revert AmountIsZero();
    // ...
    uint256 fees = IRouterClient(CCIP_ROUTER).getFee(_destinationChainSelector, evm2AnyMessage);

    // Check ETH balance for fees
    if (fees > address(this).balance) revert NotEnoughBalance(address(this).balance, fees);

    // Approve the Router to spend tokens
    IERC20(_token).approve(CCIP_ROUTER, _amount);

    // Pay fees with ether
    messageId = IRouterClient(CCIP_ROUTER).ccipSend{value: fees}(_destinationChainSelector, evm2AnyMessage);
    // ...
}
```

```
/// @notice Retries a failed message using only the messageId
/// @param messageId The ID of the failed message to retry
/// @dev This function is payable but doesn't use msg.value - fees are paid by the contract's ETH balance
function retryFailedMessage(bytes32 messageId) external payable nonReentrant {
    // Verify message and mark as resolved
    Client.Any2EVMMessage memory message = _verifyMessageFailedAndMarkResolved(messageId);

    // Process the message
    _processMessageInternal(message);

    emit MessageRecovered(messageId);
}
```

If there are conditions in `ccipReceive()` or Maple integration that cause certain messages to fail consistently, an adversary can drive repeated processing attempts or cause many messages that require retries, leading the contract to act as a gas station without having configured fees that compensate for the ETH being spent, and the current setters do not enforce any relationship between configured fees and worst-case CCIP or retry costs.

Recommendations

Strengthen the fee configuration and validation model so that deposit and redemption fees are sized with respect to the CCIP and retry costs that the contract may incur, for example by validating fee levels against `estimateFee()` for supported chains.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Fee validation is managed by Maple off-chain."*

CPU-10	Insufficient Balance Check In `withdrawNative()`
Asset	`CCIPUniversalUpgradeable.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `withdrawNative()` function only checks that the contract ETH balance is non-zero before attempting to send the requested `amount`, but it does not validate that the balance is at least `amount`. If the requested `amount` exceeds the current balance, the low-level call will fail and revert without a clear, protocol specific error that indicates an insufficient balance condition.

```
function withdrawNative(uint256 amount, address payable to) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (to == address(0)) revert CCIPUniversalInvalidParameters();
    if (address(this).balance == 0) revert NothingToWithdraw();
    (bool success, ) = to.call{value: amount}("");
    if (!success) revert FailedToWithdrawEth(address(this), to, amount);
    emit EthWithdrawn(to, amount);
}
```

Recommendations

Add an explicit validation that the contract balance is greater than or equal to the requested `amount` and revert with a clear custom error when there is insufficient ETH to perform the withdrawal.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Admin-only function with acceptable revert behavior."*

CPU-11	Ambiguous Fee Semantics In `getCollectedFees()`
Asset	`CCIPUniversalUpgradeable.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `getCollectedFees()` view function exposes `s_tokenFeesBalances[token]` with documentation that implies it returns the accumulated token amount collected as fees for a specific token, but in practice this storage is used for both asset-denominated deposit fees and share-denominated redemption fees. This mixes two different fee semantics (underlying tokens versus pool shares) behind a single getter and Natspec description, which can mislead callers about what the returned value represents in each context.

```
/// @notice Get the collected balance for a specific token
/// @param token The token address
/// @return The accumulated token amount collected as fees
function getCollectedFees(address token) external view returns (uint256) {
    return _getCCIPUniversalStorage().s_tokenFeesBalances[token];
}
```

Recommendations

Clarify and document the semantics of collected fees for both deposit and redemption flows, for example by distinguishing between asset-denominated and share-denominated balances in the storage layout and Natspec, or by introducing separate views so that callers understand exactly which type of fee and unit each getter is exposing.

Resolution

Documentation was updated to clarify return value semantics in commit 680c301.

CPU-12	Failed Messages Getter `getFailedMessages()` Returns Resolved Entries
Asset	`CCIPUniversalUpgradeable.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `getFailedMessages()` view function returns a mixture of both failed and resolved messages, even though the name and documentation suggest that it returns only failed messages. This can mislead offchain tooling or operators that rely on this function to enumerate messages that are still in a failed state.

The storage layout and error code enum show that `s_failedMessages` is a generic state map for messages, not a set of only failed entries:

```
struct CCIPUniversalStorage {
    // The message contents of failed messages are stored here for token recovery.
    mapping(bytes32 => Client.Any2EVMMessage) s_messageContents;
    // Contains failed messages and their state.
    EnumerableMap.Bytes32ToUintMap s_failedMessages;
    // ...
}

enum ErrorCode {
    RESOLVED,
    FAILED
}
```

However, `getFailedMessages()` simply paginates over `s_failedMessages` without filtering by `ErrorCode.FAILED`:

```
function getFailedMessages(uint256 offset, uint256 limit) external view returns (FailedMessage[] memory) {
    CCIPUniversalStorage storage $ = _getCCIPUniversalStorage();
    uint256 length = $.s_failedMessages.length();

    if (offset >= length) {
        return new FailedMessage[](0);
    }

    uint256 returnLength = (offset + limit > length) ? length - offset : limit;
    FailedMessage[] memory failedMessages = new FailedMessage[](returnLength);

    for (uint256 i = 0; i < returnLength; i++) {
        (bytes32 messageId, uint256 errorCode) = $.s_failedMessages.at(offset + i);
        failedMessages[i] = FailedMessage(messageId, ErrorCode(errorCode));
    }

    return failedMessages;
}
```

As a result, callers that assume all returned entries are actively failed messages may misinterpret the state and, for example, repeatedly treat already resolved messages as still pending failure. Other onchain functions that perform recovery correctly validate that the status is `ErrorCode.FAILED` before acting, so this issue does not directly lead to loss of funds, but it degrades observability and can cause confusing or brittle offchain integrations.

Recommendations

Either change the implementation of `getFailedMessages()` to filter for entries that are still in the `ErrorCode.FAILED` state, or clearly separate the views so that one function returns only failed messages and another returns all messages with their states.

Resolution

Function now correctly filters to return only `ErrorCode.FAILED` messages, matching the function name and documentation.

Fixed in commit 680c301.

CPU-13	Interface Inconsistency In `removeSharesById()` Return Values
Asset	`Interfaces.sol`
Status	**Resolved:**
Rating	Informational

Description

The local interface for the withdrawal manager omits the `sharesRemaining` return value from `removeSharesById()`, which diverges from the Maple withdrawal manager implementation.

In `MapleWithdrawalManager.sol`, `removeSharesById()` returns both the number of shares returned and the number of shares remaining in the request:

```
function removeSharesById(
    uint256 requestId_,
    uint256 sharesToRemove_
)
    external override whenProtocolNotPaused nonReentrant
    returns (uint256 sharesReturned_, uint256 sharesRemaining_)
{ ... }
```

However, in `Interfaces.sol`, the interface exposes only a single return value, and `CCIPUniversalUpgradeable.sol` consumes it as such, discarding the remaining shares:

```
function removeSharesById(uint256 requestId, uint256 sharesToRemove)
    external
    returns (uint256 sharesReturned);
```

Recommendations

Update `IWithdrawalManagerLike.removeSharesById()` to match the Maple withdrawal manager signature and include the `sharesRemaining` return value so callers have access to the full interface and consistent ABI expectations.

Resolution

Fixed in commit 680c301

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

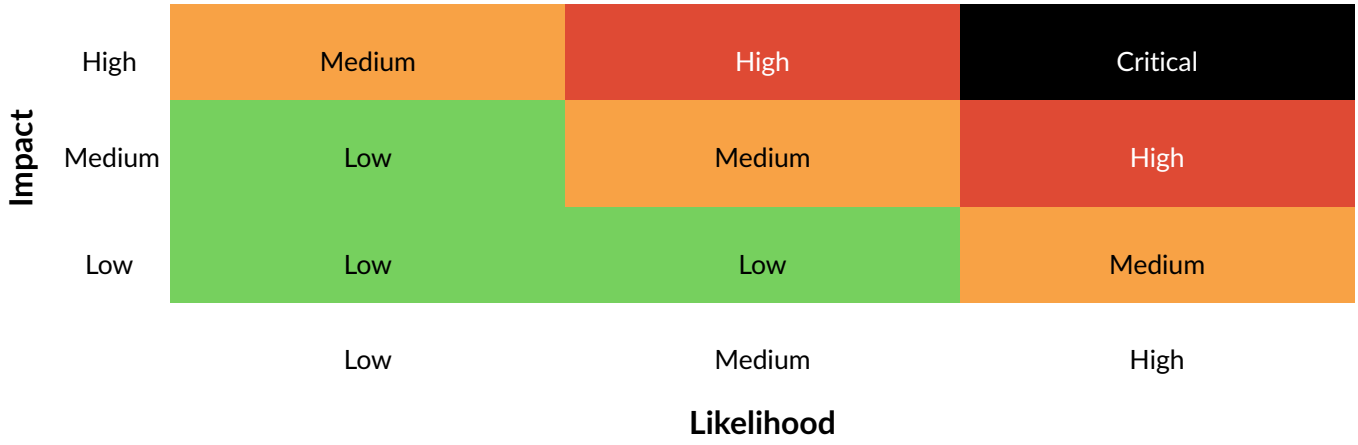


Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].

