SHERLOCK

Security Review For Rain

Private Best Efforts Contest Prepared For:	Rain
Lead Security Expert:	hash
Date Audited:	December 2 - December 8, 2024

Introduction

Rain, a Visa Network issuer, provides solutions for card programs tailored to various regions and use cases. Users can leverage their collateral contract balance to use Visa credit cards. This contest focuses on how Rain and its users can efficiently manage collateral and on-chain statements.

Scope

Repository: SignifyHQ/collateral-contract-evm-v2

Branch: main

Audited Commit: b432939abb6b322545a71f448f69b6db73d4df44

Final Commit: 31dd0b06b37a21d20535875ba9d47cc9d6e5e9e9

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
0	2

Issues Not Fixed or Acknowledged

High	Medium
0	0

Security experts who found valid issues

hash	0xc0ffEE	aslanbek

Issue M-1: Users can pay off statements by just using 50% of the owed value

Source: https://github.com/sherlock-audit/2024-11-rain-judging/issues/9

Found by

0xc0ffEE, aslanbek

Summary

The rounding half up in the function `Coordinator::getAssetAmountCents()` can allow users to pay off all owed value by just using 50% of that value

Root Cause

- The function Coordinator::getAssetAmountCents() converts the asset amount to cents, which rounds half up the result. By this, the `amountCents` could be deviated by maximum `0.5cents` from the actual amount.

```
    function getAssetAmountCents(
        address _asset,
        uint256 _amountNative
    ) public view returns (uint256 amountCents, uint8 assetDecimals) {
        // Check if asset is supported
        if (!isSupportedAsset(_asset)) {
            revert UnsupportedAsset(_asset);
        }

        SupportedAsset memory asset = supportedAssets[_asset];
        assetDecimals = _asset == address(0)
            ? 18 // All EVM chains should implement 18 decimals in its native asset
            : IERC20Decimals(_asset).decimals();

        if (asset.oracle == address(0)) {
            // asset is considered to be stable to USD
            uint256 divisor = 10 ** (assetDecimals - 2);
            // Add half of the divisor to round
@>              amountCents = (_amountNative + (divisor / 2)) / divisor;
        } else {
            (int256 price, uint8 priceDecimals) = _getCurrentAssetPrice(asset);
            uint256 divisor = 10 **
                (uint256(priceDecimals) - 2 + assetDecimals);
@>              amountCents =
                // Add half of the divisor to round
                ((_amountNative * uint256(price)) + (divisor / 2)) /
                divisor;
        }
    }
```

- The function allows an arbitrary user to pay for a statement given that the asset is supported. The asset amount is converted to cents using the function `getAssetAmountCents()` above.

```
    function makePaymentFromUserAccountForStatement(
        address _asset,
        uint256 _amountNative,
        string calldata _statementId
    ) external payable {
        // Transfer asset from user account to treasury
        (
            uint256 amountCents,
            uint256 feeNative,
            uint8 assetDecimals
@>        ) = _makePaymentFromUserAccount(_asset, _amountNative);
@>        _markStatementPaid(statements[_statementId], amountCents);

        emit PaymentFromUserAccountForStatement(
            msg.sender,
            _asset,
            _amountNative,
            amountCents,
            feeNative,
            assetDecimals,
            statements[_statementId].teamId,
            _statementId
        );
    }

    function _makePaymentFromUserAccount(
        address _asset,
        uint256 _amountNative
    )
        internal
        nonReentrant
        onlySupportedAsset(_asset)
        returns (uint256 amountCents, uint256 feeNative, uint8 assetDecimals)
    {
        // Calculate fee with proper rounding
        feeNative =
            ((_amountNative * supportedAssets[_asset].feeBps) + 5000) /
            10000;
        uint256 amountWithFeeNative = _amountNative + feeNative;
        // Convert amount to cents
@>          (amountCents, assetDecimals) = getAssetAmountCents(
            _asset,
            _amountNative
        );
    ...
    }
```

So far, by paying `0.5cents` value of the asset, there will be `1cent` owed amount is deducted from the statement => the users can pay the full owed amount by paying 0.5 cents each time. Even though the gas needed to pay for the full owed amount is high but it is affordable for many networks with cheap gas fee. Note that, the same issue also happens with the function `makePaymentFromCollateralForStatement()`, `makePaymentFromCollateral()`, `makePaymentFromUserAccount()`

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. A statement is created with 100 USD owed amount, equivalent `10000` cents
2. A user makes `10000` calls to function `makePaymentFromUserAccountForStatement()`, in each call the asset value equals to `0.5cents`. As a result, the user only pays `5000cents`

Impact

- 50% of statements values is loss

PoC

Add this test to the file `test/payments.ts`

```
it.only("pay half cent", async function () {
  // @audit pay half cent
  const oneHourAfter = Math.floor(new Date().getTime() / 1000) + 60 * 60;
  await coordinator.connect(publisher).updateStatement(
    statementId,
    teamId,
    10000n, // 100 usd
    collateralProxyAddress,
    oneHourAfter,
  );
  let statement = await coordinator.statements(statementId);

  const user1BalanceBefore = await stableAsset1.balanceOf(user1Address);

  await stableAsset1
    .connect(user1)
    .approve(coordinatorAddress, user1BalanceBefore);

  for (let i = 0; i < 10000; ++i) {
    await coordinator.connect(user1).makePaymentFromUserAccountForStatement(
      stableAsset1Address,
      oneDollarStableAsset / 200n, // 0.5 cent
      statementId,
    );
  }

  statement = await coordinator.statements(statementId);
  expect(statement[3]).to.eq(10000n); // equal to 10000 cents
});
```

Run the test and it succeeds.

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/SignifyHQ/collateral-contract-evm-v2/pull/26

Issue M-2: min/maxAnswer check is not done in for Chainlink oracles

Source: https://github.com/sherlock-audit/2024-11-rain-judging/issues/47

Found by

hash

Summary

Not validating for min/max answer on chainlink feeds can cause loss of funds for the credit issuer

Root Cause

When reading from Chainlink feeds validation is not done for min/max answers even though such oracles are used.

Eg: ETH/USD chainlink feed in optimism has min and max value set

Internal pre-conditions

No response

External pre-conditions

Huge crash in price of tokens should occur

Attack Path

1. Price of token A drops below the minAnswer set on the feed (eg: 100 is the minAnswer and the token price goes to 1)
2. A statement is published with amount == 100
3. Attacker pays the statement balance by transferring 1 token A (incorrectly priced at 100 instead of 1) and the statement is settled
4. Loss for the credit issuer since they only receive 1 value compared to 100

Impact

Loss of funds for the credit issuer

PoC

No response

Mitigation

Check for the min/maxAnswer on the feeds and disallow payments using that token if the these thresholds are met

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/SignifyHQ/collateral-contract-evm-v2/pull/29

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.