SHERLOCK

Security Review For Townsquare

Collaborative Audit Prepared For: **Townsquare**
Lead Security Expert(s): **A2-security**
Date Audited: **September 26 - October 4, 2025**

Introduction

TownSquare is a modular crosschain lending market protocol that allows low-latency single chain and crosschain lending/borrowing. This is an audit over the primary lending and borrowing functionalities in single hub chain and crosschain contexts.

Scope

Repository: TowneSquare/ts-crosschain-contracts

Audited Commit: d0055612feb9cb5e4b6ca1305479dde431e7178f

Final Commit: 1d9d6fc1d271d85dcc471780335574cc7f0116ea

Files:

- contracts/bridge/BridgeMsg.sol
- contracts/bridge/HubRouter.sol
- contracts/bridge/BridgeRouter.sol
- contracts/bridge/SpokeRouter.sol
- contracts/bridge/CCIPAdapter.sol
- contracts/bridge/CCIPDataAdapter.sol
- contracts/bridge/CCIPTokenAdapter.sol
- contracts/bridge/MainStateAdapter.sol
- contracts/bridge/LayerZeroAdapter.sol
- contracts/bridge/libraries/CCIPMsg.sol
- contracts/bridge/libraries/CCTPMsg.sol
- contracts/bridge/libraries/Messages.sol
- contracts/hub/AccountManager.sol
- contracts/hub/HubCircleTokenPool.sol
- contracts/hub/HubNonBridgedTokenPool.sol
- contracts/hub/HubPool.sol
- contracts/hub/HubPoolState.sol
- contracts/hub/Hub.sol

- contracts/hub/interfaces/AccountManager.sol
- contracts/hub/libraries/DataTypes.sol
- contracts/hub/libraries/Utils.sol
- contracts/hub/LoanManager.sol
- contracts/hub/LoanManagerState.sol
- contracts/hub/logic/PoolLogic.sol
- contracts/hub/logic/LiquidationLogic.sol
- contracts/hub/logic/LoanManagerLogic.sol
- contracts/hub/logic/LoanPoolLogic.sol
- contracts/hub/logic/RewardLogic.sol
- contracts/hub/logic/UserLoanLogic.sol
- contracts/hub/OracleManager.sol
- contracts/hub-rewards/HubRewards.sol
- contracts/hub/SpokeManager.sol
- contracts/oracle/nodes/PriceDeviationCircuitBreakerNode.sol
- contracts/oracle/nodes/PriceDeviationSameOracleCircuitBreakerNode.sol
- contracts/oracle/nodes/PythNode.sol
- contracts/oracle/nodes/RedStoneNode.sol
- contracts/oracle/nodes/ReducerNode.sol
- contracts/oracle/nodes/StalenessCircuitBreakerNode.sol
- contracts/oracle/nodes/VaultAssetToSharesNode.sol
- contracts/oracle/storage/FeedType.sol
- contracts/oracle/storage/NodeDefinition.sol
- contracts/oracle/storage/NodeOutput.sol
- contracts/spoke/AlwaysEligibleAddressOracle.sol
- contracts/spoke/RateLimited.sol
- contracts/spoke-rewards/SpokeRewardsV2Common.sol

- contracts/spoke-rewards/SpokeRewardsV2Erc20Token.sol
- contracts/spoke-rewards/SpokeRewardsV2GasToken.sol
- contracts/spoke-rewards/SpokeRewardsV2Token.sol
- contracts/spoke/SpokeCircleToken.sol
- contracts/spoke/SpokeCommon.sol
- contracts/spoke/SpokeErc20Token.sol
- contracts/spoke/SpokeGasToken.sol
- contracts/spoke/SpokeState.sol
- contracts/spoke/SpokeToken.sol

Final Commit Hash

1d9d6fc1d271d85dcc4717803355574cc7f0116ea

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- 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.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
1	4	0

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: LayerZero under-delivery of msg.value enables value withholding [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/issues/63

Summary

`LayerZeroAdapter` lets delivery happen with less `msg.value` than quoted. `receiverValue` is priced via options but not enforced on-chain; anyone can call the endpoint's `lzReceive` and forward whatever `msg.value` they send (even 0).

Vulnerability Detail

In send, the adapter includes `receiverValue` in options and fee quote:

```
bytes memory options = OptionsBuilder
    .newOptions()
    .addExecutorLzReceiveOption(message.params.gasLimit.toUint128(),
    ↪  message.params.receiverValue.toUint128())
    .addExecutorLzComposeOption(0, message.params.gasLimit.toUint128(), 0);

Messages.MessagePayload memory messagePayload =
↪  Messages.decodeActionPayload(message.payload);

// send using layerZero ENdpoint
MessagingReceipt memory receipt = endpoint.send{ value: msg.value }(
    MessagingParams(lzChainId, adapterAddress, payloadWithMetadata, options, false),
    Messages.convertGenericAddressToEVMAddress(messagePayload.userAddress)
);
```

But `receiverValue` is not part of the message payload/metadata used on destination.

LayerZero endpoint executes via a public payable function and forwards whatever `msg.value` is supplied

Result: executors can deliver with `msg.value = 0` while fees were quoted including `receiverValue`. Even if the executor is honest, anyone can front-run and call `lzReceive` first with 0 value.

Impact

It is possible to deliver the LayerZero message with lower native fees than intended.

Code Snippet

https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/blob/e76835dd56fe6afaf6fe842f69b24203eba8250b/ts-crosschain-contracts/contracts/bridge/LayerZeroAdapter.sol#L58-L69

Tool Used

Manual Review

Recommendation

Simplest safe fix without changing shared `Messages` structs: block messages that require a nonzero `receiverValue` and set LayerZero options to zero.

```
  function sendMsg(Msg.MsgToSend memory message) external payable
   ↪  override onlyBridgeRouter {
    (uint32 lzChainId, bytes32 adapterAddress) =
     ↪  getChainAdapter(message.destinationChainId);

+   // Prevent msg.value under-delivery; LayerZero does not enforce receiverValue
 ↪  on receive
+   if (message.params.receiverValue > 0) revert UnsupportedReceiverValue();

    // ... existing code
```

Issue M-1: Non-EVM sender decode reverts in CCIP adapter [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/issues/62

Summary

CCIP adapters decode `message.sender` as an EVM `address`. For non-EVM chains with sender encodings not limited to 20 bytes, this decode reverts, making messages permanently undeliverable.

Vulnerability Detail

The receive handlers assume `Client.Any2EVMMessage.sender` encodes an EVM `address`:

```
// check source chain and source address
uint16 townSqChainId = ccipChainIdTotownSqChainId[message.sourceChainSelector];
bytes32 sourceAddress =
↪  Messages.convertEVMAddressToGenericAddress(abi.decode(message.sender,
↪  (address)));
(uint64 ccipChainId, bytes32 adapterAddress) = getChainAdapter(townSqChainId);
if (message.sourceChainSelector != ccipChainId) revert
↪  ChainUnavailable(townSqChainId);
```

But `sender` is `bytes` and may represent non-EVM addresses (e.g., 32 bytes or other sizes). Decoding with `abi.decode(...,(address))` assumes a 20-byte EVM ABI; for other formats it reverts. Example: a non-EVM chain provides a 32-byte address → decode reverts → message never delivered.

Impact

Liveness failure for messages from non-EVM chains.

Code Snippet

https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/blob/e76835dd56fe6afaf6fe842f69b24203eba8250b/ts-crosschain-contracts/contracts/bridge/CCIPDataAdapter.sol#L41-L46

Tool Used

Manual Review

Recommendation

If supporting non-EVM chains, don't always decode `sender` as `address`. Treat it as a generic address first, and only convert to EVM when safely within 160 bits:

```
- bytes32 sourceAddress =
 ↪  Msgs.convertEVMAddressToGenericAddress(abi.decode(msg.sender, (address)));
 ↪
+ bytes32 sourceAddress = bytes32(message.sender);
+ // non-evm addresses can also be less than 160bits, but no risk on that
+ if (uint256(sourceAddress) < type(uint160).max) {
+   sourceAddress =
 ↪  Messages.convertEVMAddressToGenericAddress(abi.decode(message.sender,
 ↪  (address)));
+ }
```

Issue M-2: Wrong sender used for LayerZero fee quote causes incorrect pricing [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/issues/65

Summary

`LayerZeroAdapter.getSendFee` uses `msg.sender` when quoting. LayerZero's endpoint derives nonce/library from the provided sender; using the router instead of the adapter yields incorrect quotes.

Vulnerability Detail

Adapter passes `msg.sender` into `endpoint.quote`:

```
MessagingFee memory messingFee = endpoint.quote(
MessagingParams(lzChainId, adapterAddress, payloadWithMetadata, options, false),
msg.sender
);
fee = messagingFee.nativeFee;
```

Endpoint `quote` uses the sender for nonce and send library selection:

```
function quote(MsgingParams calldata _params, address _sender) external view
 →
    returns (MsgingFee memory) {
    uint64 nonce = outboundNonce[_sender][_params.dstEid][_params.receiver] + 1;
    address _sendLibrary = getSendLibrary(_sender, _params.dstEid);
    return ISendLib(_sendLibrary).quote(packet, _params.options,
}    ‹    _params.payInLzToken);
```

The true sender during `send` is the adapter (`address(this)`), not the router that calls `getSendFee`.

Impact

- Over-quote: refunds, but inaccurate.
- Under-quote: insufficient fee → send reverts (DoS for that adapter path).

Code Snippet

https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/blob/e76835d

d56fe6afaf6fe842f69b24203eba8250b/ts-crosschain-contracts/contracts/bridge/LayerZeroAdapter.sol#L84-L89

Tool Used

Manual Review

Recommendation

Quote with `address(this)` to match actual sender used in `send`:

```
-        MsgingFee memory msgingFee = endpoint.quote(
-            MsgingParams(lzChainId, adapterAddress, payloadWithMetadata,
 ↪  options, false),
-            msg.sender
-        );
+        MsgingFee memory msgingFee = endpoint.quote(
+            MsgingParams(lzChainId, adapterAddress, payloadWithMetadata,
 ↪  options, false),
+            address(this)
+        );
        fee = messagingFee.nativeFee;
```

Issue M-3: RedStone TWAP loop ineffective and uses latest price only [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/issues/66

Summary

`getTwapPrice()` walks back rounds with `getRoundData(--roundId)`. RedStone push feeds keep `roundId == 1`, so the loop only tries once, fails in try/catch, and TWAP collapses to the latest price. It wastes gas and provides no real TWAP.

Vulnerability Detail

When `twapTimeInterval > 0`, the code averages historical rounds by decrementing `latestRoundId`:

RedStone push feeds report `roundId == 1` for latest data, and do not have prior rounds. The first `--latestRoundId` becomes 0 → `getRoundData(0)` reverts; the catch swallows it; the loop ends with only the latest sample. Outcome: TWAP does not smooth anything and may use stale prices.

Impact

- TWAP ineffective; averages 1 value (latest only).
- Risk of using stale prices without any freshness guard.

Code Snippet

https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/blob/e76835dd56fe6afaf6fe842f69b24203eba8250b/ts-crosschain-contracts/contracts/oracle/nodes/RedStoneNode.sol#L58-L66

Tool Used

Manual Review

Recommendation

Add a freshness check (heartbeat) and revert on stale data. If the feed does not support historical rounds, skip TWAP (or revert) instead of looping. Heartbeat can be sourced from RedStone's UI: RedStone feeds.

Example change around processing the latest round:

```
    (uint80 roundId, int256 answer, , uint256 updatedAt, ) =
    ↪  chainlinkAggregator.latestRoundData();
  + if (updatedAt + MaxInterval < block.timestamp) revert("stale price");

  -  uint256 price = twapTimeInterval == 0
  -            ? answer.toUint256()
  -            : getTwapPrice(chainlinkAggregator, roundId, answer.toUint256(),
  ↪  twapTimeInterval);
  + // If the feed lacks historical rounds (e.g., RedStone push feeds), skip TWAP
  + uint256 price = answer.toUint256();
```

Alternatively, use Chainlink feeds that provide proper round history for this TWAP implementation.

Issue M-4: Stable rate manipulation via utilisation ratio manipulation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/issues/67

Summary

Stable borrow rate is computed from the current utilisation. Attackers can temporarily change utilisation (flash-loan/temporary deposit) to lock in an abnormally low stable rate, then revert utilisation.

Vulnerability Detail

`updateInterestRates()` derives the stable rate directly from the latest utilisation:

```
uint256 totalDebt = poolAmountDataCache.variableBorrowTotalAmount +
↪  poolAmountDataCache.stableBorrowTotalAmount;
uint256 utilisationRatio = MathUtils.calcUtilisationRatio(totalDebt,
↪  poolData.depositData.totalAmount);
uint32 vr1 = poolData.variableBorrowData.vr1;

// calculate new interest rates
uint256 variableBorrowInterestRate = MathUtils.calcVariableBorrowInterestRate(
    poolData.variableBorrowData.vr0,
    vr1,
```

```
uint256 stableBorrowInterestRate = MathUtils.calcStableBorrowInterestRate(
    vr1,
    poolData.stableBorrowData.sr0,
    poolData.stableBorrowData.sr1,
    poolData.stableBorrowData.sr2,
    poolData.stableBorrowData.sr3,
    utilisationRatio,
    poolData.depositData.optimalUtilisationRatio,
    MathUtils.calcStableDebtToTotalDebtRatio(poolAmountDataCache.stableBorrowTotalA
    ↪  mount, totalDebt),
    poolData.stableBorrowData.optimalStableToTotalDebtRatio
);
```

Attack (one tx or two quick blocks):

- Deposit a large amount (or borrow/repay) to drop utilisation (e.g., 80% → 10%).
- Take stable loan at the now-low rate (e.g., 0.1% vs market 8%).
- Withdraw the temporary deposit (or finish flash-loan).

- The loan keeps the low stable rate until rebalanced.

Impact

- Borrowers can lock very low stable rates; lenders earn below-market interest. A rebalancer can fix via `rebalanceUP`, but it's manual per address and reactive.

Code Snippet

https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/blob/e76835dd56fe6afaf6fe842f69b24203eba8250b/ts-crosschain-contracts/contracts/hub/logic/HubPoolLogic.sol#L439-L446 https://github.com/sherlock-audit/2025-09-townsquare-audit-sep-2025/blob/e76835dd56fe6afaf6fe842f69b24203eba8250b/ts-crosschain-contracts/contracts/hub/logic/HubPoolLogic.sol#L451-L461

Tool Used

Manual Review

Recommendation

Cap how much the stable rate can change within one update and block large jumps within the same block. Keep the ability for `REBALANCER_ROLE` to adjust users later.

Example guard after computing `stableBorrowInterestRate`:

```
    uint256 stableBorrowInterestRate = MathUtils.calcStableBorrowInterestRate(
        vr1,
        poolData.stableBorrowData.sr0,
        poolData.stableBorrowData.sr1,
        poolData.stableBorrowData.sr2,
        poolData.stableBorrowData.sr3,
        utilisationRatio,
        poolData.depositData.optimalUtilisationRatio,
        MathUtils.calcStableDebtToTotalDebtRatio(poolAmountDataCache.stableBorrowTo
        ↪  talAmount, totalDebt),
        poolData.stableBorrowData.optimalStableToTotalDebtRatio
    );

+   // Deviation cap and same-block manipulation check
+   uint256 previousStableRate = poolData.stableBorrowData.interestRate;
+   if (previousStableRate > 0) {
+       uint256 maxDeviation = previousStableRate * 500 / 10000; // 5%
+       uint256 deviation = stableBorrowInterestRate > previousStableRate
+           ? stableBorrowInterestRate - previousStableRate
+           : previousStableRate - stableBorrowInterestRate;
```

```
+       if (deviation > maxDeviation && block.timestamp ==
 ↪  poolData.lastUpdateTimestamp) {
+           revert("stable rate jump");
+       }
+   }
```

This won't fully stop manipulation, but it prevents large same-block swings and buys time for the rebalancer.

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.