Qubit

Cross-chain Smart Contracts Security Audit

rev 1.1

Prepared for
Mound

Prepared by
Theori

December 13, 2021

Theori

Table of Contents

Executive Summary

Theori reviewed the X-Collateral (Cross-Chain Collateral) feature in Qubit. This new feature allows users to borrow assets from one chain using assets from another chain as collateral. No critical issues were identified in the current code. It may be possible for a user to use the relayer's gas to execute code, but whether it is profitable depends on the chain's gas fees and the bridge's fee. Lastly, there is the possibility of denial-of-service attacks on the relayer, and while we identified one such issue, there are likely additional denial-of-service vectors.

Scope

The scope of this review was limited to the X-Collateral feature in Qubit. It does not review prior code nor code that is part of other products.

Source code:

- mound/audit/qubit
    - 5a04432b36882cb1f73e79a71345c552158cce45

Fixes:

- mound/audit/qubit
    - de8566b172db91b045d4dec26e85e1404957a2cc

Findings

These are the potential issues that may have correctness and/or security impacts.

Summary

#	ID	Title	Severity
1	THE-XC-001	`supplyAndBorrowBehalf` does not call `_enterMarket` on the `borrowMarket`	Not Applicable
2	THE-XC-002	`supplyAndBorrowBNB` calls user contract with relayer's gas	High
3	THE-XC-003	Deposit with invalid `destinationDomainID` causes denial of service in relayer	High
4	THE-XC-004	Deposit allows passing invalid option	Low
5	THE-XC-005	`supplyAndBorrowBehalf` and `supplyAndBorrowBNB` should pass `uAmount` to `_borrowAllowed`	Low

Issue #1: supplyAndBorrowBehalf does not call _enterMarket on the borrowMarket

ID	Summary	Severity
THE-XC-001	*supplyAndBorrowBehalf* calls *_enterMarket* on the *supplyMarket*, but not on the *borrowMarket*, resulting in incorrect calculations of the user's debt and allowing them to steal funds	Not Applicable

During the audit of Qubit, it was established that *_enterMarket* must be called for a market if the user borrows from that market. Otherwise, the debt will not be included in *_getAccountLiquidityInternal* and it would be possible for the user to withdraw their collateral without repaying the debt.

In Qore.sol, *supplyAndBorrowBehalf* does not call *_enterMarket* on *borrowMarket*. While *supplyAndBorrowBehalf* is not currently used, it is a critical issue that should be addressed if this function will be used in the future.

Recommendations

Add a call to *_enterMarket* for *borrowMarket*.

```
function supplyAndBorrowBehalf(address account, address supplyMarket, uint
supplyAmount, address borrowMarket, uint borrowAmount)
    external
    payable
    override
    onlyListedMarket(supplyMarket)
    onlyListedMarket(borrowMarket)
    onlyWhitelisted
    nonReentrant
    returns (uint)
{
    address underlying = IQToken(supplyMarket).underlying();
    uint uAmount = underlying == address(WBNB) ? msg.value : supplyAmount;

    uint qAmount = IQToken(supplyMarket).supplyBehalf{ value: msg.value }(msg.sender,
account, uAmount);

    _enterMarket(supplyMarket, account);
    _enterMarket(borrowMarket, account);
```

```
    require(_borrowAllowed(supplyMarket, supplyAmount, borrowMarket, borrowAmount),
"Qore: cannot borrow");
    IQToken(borrowMarket).borrow(account, borrowAmount);

    qDistributor.notifySupplyUpdated(supplyMarket, account);
    qDistributor.notifyBorrowUpdated(borrowMarket, account);
    return qAmount;
}
```

Fix

Mound removed the *supplyAndBorrowBehalf* function.

Issue #2: supplyAndBorrowBNB calls user contract with relayer's gas

ID	Summary	Severity
THE-XC-002	Relayers use a dynamic gas limit when calling *executeProposal*, which may call a user's contract, allowing users to steal gas from the relayer	High

In QToken.sol, *_doTransferOut* calls *SafeToken.safeTransferETH(to, amount)* if *underlying* is WBNB, which executes a call to address *to* with no gas limit. This code runs when borrowing qBNB, e.g., from *supplyAndBorrowBNB*, which is called from the bridge. The relayer uses a dynamic gas limit when calling *executeProposal*, allowing users to steal arbitrary amounts of gas from the relayer.

```
let estimatedGas = (await method.estimateGas({ from: relayer }))*1.2
estimatedGas = parseInt(estimatedGas.toString())
const txReceipt: TransactionReceipt = await method.send({ from: relayer, gas:
estimatedGas, gasPrice: gasPrice.toString(), nonce: nonce.toString() })
```

The user could profit from this free gas depending on the gas economics of the blockchain. For example, they could mint and sell large amounts of CHI (or another gas token). Or they may use it to spam the relayer with calls to expensive contracts and cause a denial-of-service.

Recommendations

Relayers should use a fixed gas limit when calling *executeProposal* to limit the gas used by a user's contract. Additionally, the deposit fee should be set high enough to ensure users cannot profit from the amount of gas they can use.

Fix

Mound set the estimated gas to a fixed value in the relayer for the Ethereum and BSC chains.

```
if (this.config.type == ChainType.Ethereum) {
    estimatedGas = 200000
} else if (this.config.type == ChainType.BinanceSmart) {
    estimatedGas = 600000
} else {
    estimatedGas = (await method.estimateGas({ from: relayer })) * 3
```

```
}
```

Issue #3: Deposit with invalid destinationDomainID causes denial of service in relayer

ID	Summary	Severity
THE-XC-003	The relayer throws an error when handling a deposit with an unknown *destinationDomainID*, preventing processing of future deposits	High

When a relayer sees an event with an unknown *domainID*, it throws an exception. Because no state was updated to prevent handling this event, future cron jobs will experience the same error, causing denial of service to all deposits after the invalid one.

```
for (let each of events) {
    if (!router[each.destId]) {
        throw new Error(`[Chain${this.config.domainId}] Route Failed - src:
${each.srcId}, dest: ${each.destId}, nonce: ${each.nonce}`)
    }
    await router[each.destId].handleProposal(each)
}
```

Recommendations

Rather than throw an exception, the relayer should log invalid deposits in the firebase database along with the cancelled and failed proposals.

Fix

Mound replaced throwing an error with logging an error.

```
for (let each of events) {
    if (!router[each.destId]) {
        if (this.useMetric) {
            console.log(`[Chain${this.config.domainId}] Route Failed - src:
${each.srcId}, dest: ${each.destId}, nonce: ${each.nonce}`)
            await metricRouteFailed(this.connector.getSignerAddress(),
this.getProposalDataHash(each), each)
        }
    } else {
        await router[each.destId].handleProposal(each)
    }
}
```

Issue #4: Deposit allows passing invalid option

ID	Summary	Severity
THE-XC-004	*QBridgeHandler* does not validate that the *option* parameter is one of the expected values, potentially allowing users to bypass minimum deposit amounts	Low

In QBridgeHandler.sol, the deposit *option* parameter is decoded from the option data. The code appears to assume this is one of the four constant values, but there are no checks to enforce this assumption.

By passing an unhandled option, users will not have to provide a minimum deposit amount, because *minAmounts[resourceID][option]* will be 0 in the following require statement:

```
function deposit(bytes32 resourceID, address depositer, bytes calldata data) external
override onlyBridge {
    uint option;
    uint amount;
    (option, amount) = abi.decode(data, (uint, uint));

    address tokenAddress = resourceIDToTokenContractAddress[resourceID];
    require(contractWhitelist[tokenAddress], "provided tokenAddress is not
whitelisted");

    if (burnList[tokenAddress]) {
        require(amount >= withdrawalFees[resourceID], "less than withdrawal fee");
        QBridgeToken(tokenAddress).burnFrom(depositer, amount);
    } else {
        require(amount >= minAmounts[resourceID][option], "less than minimum amount");
        tokenAddress.safeTransferFrom(depositer, address(this), amount);
    }
}
```

Recommendations

The option parameter should be validated. This could be done either manually or via an enum type. For example, the following enum type could be used in place of *uint* for option values:

```
enum DepositOption {
    QUBIT_BNB_NONE,
    QUBIT_BNB_0100,
```

```
    QUBIT_BNB_0050,
    BUNNY_XLP_0150
}
```

can be decoded as

```
DepositOption option;
uint amount;
(option, amount) = abi.decode(data, (DepositOption, uint));
```

This would automatically insert runtime checks that the value is in the range of the enum. Note that this would change the values of these options.

If using unhandled options with no minimum deposit amount is intended behavior, then no fixes are necessary.

Fix

Mound responded that this is intended behavior.

Issue #5: supplyAndBorrowBehalf and supplyAndBorrowBNB should pass uAmount to _borrowAllowed

ID	Summary	Severity
THE-XC-005	In *supplyAndBorrowBNB*, *supplyAmount* is passed to *_borrowAllowed* even when *supplyAmount != uAmount*, allowing whitelisted callers to borrow more than should be allowed.	Low

In Qore.sol, *supplyAndBorrowBehalf* and *supplyAndBorrowBNB* deposit *supplyAmount* collateral with *supplyBehalf* and call *_borrowAllowed* to verify that collateral can support the *borrowAmount*. If BNB is supplied, then *supplyAmount* is ignored in the call to *supplyBehalf* and *msg.value* is used instead. However, *supplyAmount* is still used in the call to *_borrowAllowed* which is incorrect.

```
function supplyAndBorrowBNB(address account, address supplyMarket, uint supplyAmount,
uint borrowAmount)
    external
    payable
    override
    onlyListedMarket(supplyMarket)
    onlyWhitelisted
    nonReentrant
    returns (uint)
{
    require(borrowAmount <= 5e16, "exceed maximum amount");
    address underlying = IQToken(supplyMarket).underlying();
    uint uAmount = underlying == address(WBNB) ? msg.value : supplyAmount;
    uint qAmount = IQToken(supplyMarket).supplyBehalf{ value: msg.value }(msg.sender,
account, uAmount);

    _enterMarket(supplyMarket, account);

    address qBNB = 0xbE1B5D17777565D67A5D2793f879aBF59Ae5D351;
    _enterMarket(qBNB, account);

    require(_borrowAllowed(supplyMarket, supplyAmount, qBNB, borrowAmount), "Qore:
cannot borrow");
    IQToken(qBNB).borrow(account, borrowAmount);
    return qAmount;
}
```

Since this function can only be called by a whitelisted address, the impact of this issue is minimal. However, it demonstrates the risks of duplicating the logic of *QValidator.borrowAllowed* elsewhere.

Recommendations

Pass *uAmount* to *_borrowAllowed* instead of *supplyAmount*.

```
require(_borrowAllowed(supplyMarket, uAmount, qBNB, borrowAmount), "Qore: cannot
borrow");
```

Fix

Mound changed the argument to *_borrowAllowed* from *supplyAmount* to *uAmount*.

```
require(_borrowAllowed(supplyMarket, uAmount, qBNB, borrowAmount), "Qore: cannot
borrow");
```

Recommendations

These are the recommendations to improve the code quality for better readability, optimization, and security. They do not impose any immediate security impacts.

Summary

#	Title	Type	Importance
1	Merge `supplyAndBorrowBehalf` and `supplyAndBorrowBNB`	Code Quality	Minor
2	`supplyBehalf` should require amount transferred is the expected amount	Code Quality	Minor

Recommendation #1: Merge supplyAndBorrowBehalf and supplyAndBorrowBNB

In Qore.sol, *supplyAndBorrowBehalf* and *supplyAndBorrowBNB* implement nearly the same exact functionality. As demonstrated by the issues above, this can lead to mistakes in the implementations. Additionally, when issues are fixed, they must now be fixed in multiple places. We suggest that these functions are combined into one function, or at least their core functionality is moved into a common function.

Fix

Mound removed the *supplyAndBorrowBehalf* function.

Recommendation #2: supplyBehalf should require amount transferred is the expected amount

In QToken.sol, *supplyBehalf* transfers in assets with *_doTransferIn* and correctly uses the returned amount as the supplied amount. However, the callers of *supplyBehalf* assume that all of the tokens have been transferred and do not know the actual amount transferred. Either *supplyBehalf* should returned the amount transferred or it should require that *_doTransferIn* returns the expected amount.

Fix

Mound added an additional check to *supplyBehalf*.

```
uint transferred = _doTransferIn(sender, uAmount);
require(transferred == uAmount, "QToken: transfer amount differs");
```