Folks Finance X-Chain

Security Assessment

May 6th, 2024 — Prepared by OtterSec

Robert Chen r@osec.io

Woosun Song procfs@osec.io

Matteo Olivia matt@osec.io

Table of Contents

01 — Executive Summary

Overview

Folks Finance engaged OtterSec to assess the X-Chain program. This assessment was conducted between April 12th and May 1st, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 4 findings throughout this audit engagement.

In particular, we identified a critical oversight in the price oracle which failed to account for ERC-20 decimals (OS-FFX-ADV-00), which could potentially lead to the theft of funds. In addition, we identified the usage of a stale borrow balance in liquidation (OS-FFX-ADV-01) which prevented full liquidation.

We also recommended implementing access control on bridging methods (OS-FFX-SUG-00). The current bridging method, being permissionless, allowed for user-specified sender fields and could be exploited to deplete gas deposits of other users. Furthermore, we advised against using `tx.origin` (OS-FFX-SUG-01) to ensure compliance with ERC-4337 and to maintain proper semantics during smart contract interactions.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/blockchain-italia/ff-xchain-mart-contracts. This audit was performed against commit `607a51`.

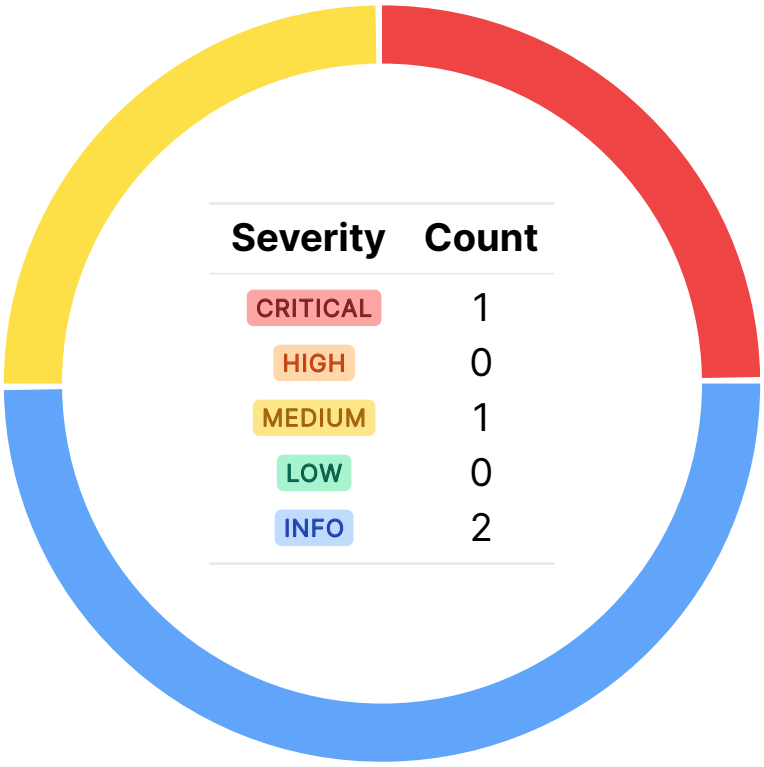
A brief description of the programs is as follows:

Name	Description
XChain-Contracts	XChain is a cross-chain lending protocol operating on a hub-spoke model. It facilitates cross-chain transactions, including deposits, withdrawals, and borrowing, while restricting liquidations to the hub chain only. The protocol leverages cross-chain communication and asset transfer primitives such as Wormhole, CCTP, and CCIP.

03 — Findings

Overall, we reported 4 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	0
MEDIUM	1
LOW	0
INFO	2

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-FFX-ADV-00	CRITICAL	RESOLVED	The price oracle does not account for ERC-20 decimals, resulting in incorrect pricing of debt and collateral assets.
OS-FFX-ADV-01	MEDIUM	RESOLVED	The liquidation logic does not account for borrow interest accrued in the violator debt.

Missing Decimal Correction CRITICAL OS-FFX-ADV-00

Description

The price oracle does not account for ERC-20 decimals, resulting in the incorrect pricing of debt and collateral assets. This occurs because the price of ERC-20 tokens is typically denoted in USD per unit (`10 ** decimals` wei) Consequently, when comparing the value of two ERC-20 tokens with different decimals, it is imperative to divide the multiplication result by `10 ** decimals` to account for the difference.

This oversight has critical security implications, as it can be easily exploited to steal funds. To clarify, we illustrate the following scenario:

1. Assume a borrowing scenario where the collateral is DAI and the debt is USDC. Note that DAI has 18 decimals, while USDC has 6 decimals. Also, assume a collateral factor of 0.4 and a borrow factor of 1, resulting in a Loan-to-Value (LTV) ratio of 40%.
2. A user deposits 1 DAI, which has a notional value of 1 USD. Thus, the user should be able to borrow at most 0.4 USDC under correct calculations.
3. However, due to the oversight, the user may actually borrow 4×10^{11} USDC. This erroneous calculation occurs because the value of 1 DAI is improperly calculated as 1×10^{18} (reflecting its full atomic units), while the value of 4×10^{11} USDC is calculated as $4 \times 10^{11} \times 10^{6} = 0.4 \times 10^{18}$.

The failure to properly account for the decimal difference leads to the collateral being critically overvalued, enabling the user to borrow vastly more than what the collateral should allow.

Remediation

Divide the price with `10**decimals` to account for varying decimals.

Patch

Fixed in 333bcf7.

Overlooked Borrow Interest MEDIUM OS-FFX-ADV-01

Description

The liquidation logic does not account for borrow interest accrued in the violator debt, resulting in the debt notional amount to be underestimated. This happens because the `calcLiquidationAmounts` function references the `violatorLoanBorrow.balance` variable without updating the borrow index.

>_ *contracts/hub/logic/LiquidationLogic.sol* solidity

```
function calcLiquidationAmounts(
    DataTypes.LiquidationLoansParams memory loansParams,
    mapping(bytes32 => LoanManagerState.UserLoan) storage userLoans,
    mapping(uint16 => LoanManagerState.LoanType) storage loanTypes,
    IHubPool collPool,
    IOracleManager oracleManager,
    uint256 maxRepayBorrowValue,
    uint256 maxAmountToRepay
) external view returns (DataTypes.LiquidationAmountParams memory liquidationAmounts) {
    /* ... */

    LoanManagerState.UserLoanBorrow storage violatorLoanBorrow =
        ↪  violatorLoan.borrows[borrPoolId];

    {
        uint256 maxRepayBorrowAmount = Math.mulDiv(maxRepayBorrowValue, MathUtils.ONE_10_DP,
            ↪  borrPrice);
        repayBorrowAmount = Math.min(maxAmountToRepay, Math.min(maxRepayBorrowAmount,
            ↪  violatorLoanBorrow.balance));
    }

    /* ... */
}
```

This oversight results in an underestimated borrow balance to be propagated into `repayBorrowAmount`. The debt of a user U is managed by the tuple (B_U, I_U), where B_U denotes the balance and I_U denotes the borrow index. The actual borrow balance is computed by $\frac{B_U \times I_G}{I_U}$ where I_G denotes a global borrow index. Consequently, B_U represents the true borrow balance only when the index is up-to-date. Since this condition is not met, $I_U < I_G$ may occur and B_U may be smaller than the true borrow balance.

Despite this being a significant calculation flaw, we deemed the impact to be moderate because it does not lead to theft of funds. Instead, the primary consequence is the underestimation of `repayBorrowAmount` which prevents a user from being fully liquidated.

Remediation

Call the `updateLoanBorrowInterests` function before referencing `violatorLoanBorrow.balance`.

Patch

Fixed in e378a26.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-FFX-SUG-00	The `BridgeRouter.sendMessage` function may be called by any address, allowing an attacker to misappropriate gas deposits of other users.
OS-FFX-SUG-01	The Spoke Bridge Router utilizes `tx.origin` which hinders compatibility and robustness.

Permissionless Bridging Function

OS-FFX-SUG-00

Description

The function `BridgeRouter.sendMessage` may be called by any address, which poses a security risk by allowing an attacker to misappropriate gas deposits of other users. The `sendMessage` function is designed to store the EVm gas token (depending on the chain) for users to cover messaging fees, enabling users to reuse excess gas for future cross-chain transactions. However, a vulnerability arises because the owner of the gas, identified by the variable `userId`, can be arbitrarily specified. This flaw allows an attacker to send cross-chain messages without incurring any fees by entering a `userId` that has a balance but is not owned by the attacker.

>_ *contracts/bridge/BridgeRouter.sol* solidity

```solidity
function sendMessage(
    Messages.MessageToSend memory message
) external payable override onlyRole(MESSAGE_SENDER_ROLE) {
    // check if valid adapter and retrieve
    IBridgeAdapter adapter = getAdapter(message.params.adapterId);

    /* ... */

    // check if have sufficient funds to pay fee (can come from existing balance and/or
        ↪  msg.value)
    bytes32 userId = _getUserId(Messages.decodeActionPayload(message.payload));
    uint256 userBalance = balances[userId];
    if (msg.value + userBalance < fee) revert NotEnoughFunds(userId);

    // update user balance considering fee and msg.value
    balances[userId] = userBalance + msg.value - fee;

    // call given adapter to send message
    adapter.sendMessage{ value: fee }(message);
}
```

We evaluated the security impact of this oversight as informational because it does not lead to theft of funds. Instead, the most significant consequence would be the depletion of another user's gas deposit.

Remediation

Implement a modifier to restrict the caller of `BridgeRouter.sendMessage`.

Patch

Fixed in 2b9d378.

Usage of tx.origin

OS-FFX-SUG-01

Description

The Spoke Bridge Router utilizes `tx.origin` which hinders compatibility and robustness.

>_ *contracts/bridge/BridgeRouterSpoke.sol* solidity

```
function _getUserIdSending(Messages.MessagePayload memory) internal view override returns
    ↪  (bytes32) {
    return Messages.convertEVMAddressToGenericAddress(tx.origin);
}
```

Using `tx.origin` is incompatible with ERC-4337 and disrupts expected behavior when the spoke is interacted with by smart contracts. In the context of ERC-4337, `tx.origin` should point to the bundler address rather than the user's address. Similarly, when the spoke is invoked by a smart contract, the user ID of the sender should reflect the transaction initiator, not the smart contract itself.

Remediation

Instead of using `tx.origin`, refer to the `payload.userAddress` variable.

Patch

Fixed in 2b9d378.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.