Ax Protocol

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Shiva Shankar sh1v@osec.io

YoungJoo Lee youngjoo.lee@osec.io

Contents

01 | Executive Summary

Overview

Ax Protocol engaged OtterSec to perform an assessment of the USX contract. This assessment of the source code was conducted between January 23rd and February 11th, 2023 by 3 engineers. For more information on our auditing methodology, see Appendix B.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team to streamline patches and confirm remediation. We delivered final confirmation of the patches March 16th, 2023.

Key Findings

Over the course of this audit engagement, we produced 6 findings total.

Specifically, we identified issues related to missing bridge fee payments (OS-USX-ADV-00), vulnerability to contract initialization frontrunning (OS-USX-ADV-01), and a minor concern regarding locked ether (OS-USX-ADV-02).

Furthermore, we provided recommendations to address potential risks related to read-only reentrancy and price manipulation (OS-USX-SUG-00), suggested stricter input parameter validation (OS-USX-SUG-01), and advised on potential gas optimizations.

02 | Scope

The source code was delivered to us in a git repository at github.com/Ax-Protocol/usx/. This audit was performed against commit 9158437.

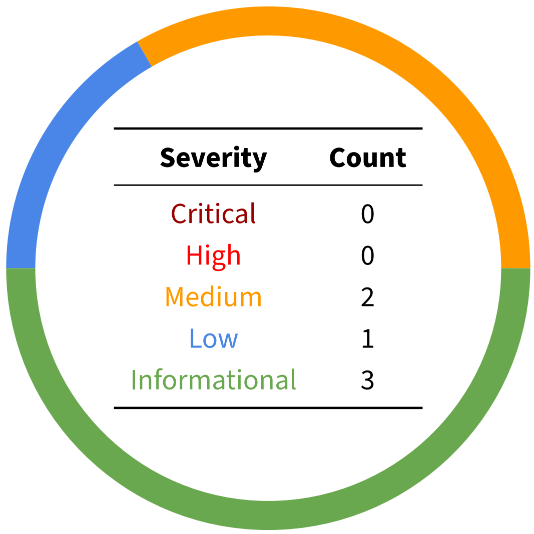
A brief description of the programs is as follows.

Name	Description
USX	USX is a cross-chain native stablecoin, built on top of LayerZero and Wormhole, that supports minting and burning by depositing and redeeming allowlisted assets respectively.

03 | Findings

Overall, we report 6 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 don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	2
Low	1
Informational	3

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-USX-ADV-00	Medium	Resolved	The absence of a sent fee during publication may result in unprocessed wormhole messages.
OS-USX-ADV-01	Medium	Resolved	Implement additional validation measures to `initialize` in `WormholeBridge.sol`, `LayerZeroBridge.sol`, and `USX.sol`.
OS-USX-ADV-02	Low	Resolved	Ether sent to `Treasury.sol` and `USX.sol` has been lost.

OS-USX-ADV-00 [med] | Missing Wormhole Bridge Fees

Description

In order to send messages using Wormhole, it is necessary to pay the bridge fee during message publication. If an incorrect fee is applied to the message call, the message publishing process will fail.

https://github.com/wormhole-foundation/wormhole/blob/main/ethereum/contracts/Implementation.sol SOLIDITY

```solidity
function publishMessage(
    uint32 nonce,
    bytes memory payload,
    uint8 consistencyLevel
) public payable returns (uint64 sequence) {
    // check fee
    require(msg.value == messageFee(), "invalid fee");

    sequence = useSequence(msg.sender);
    // emit log
    emit LogMessagePublished(msg.sender, sequence, nonce, payload,
     ↪  consistencyLevel);
}
```

As a result, the current implementation would likely abort due to invalid fees.

Remediation

Add code to pass on the entire message value to properly pay the Wormhole fees.

src/bridging/wormhole/WormholeBridge.sol DIFF

```diff
-        sequence = wormholeCoreBridge.publishMessage(0, message, 200);
+        sequence = wormholeCoreBridge.publishMessage{value:msg.value}(0,
     ↪  message, 200);
```

Consider passing on the entire message value to send ETH values as much as the message fee when `publishMessage` is called. Note that message fees can be obtained by calling `IWormhole::messageFee`.

Patch

Resolved in 777ba1d.

OS-USX-ADV-01 [med] | Unsafe Contract Initialization

Description

`initialize` in `WormholeBridge.sol`, `LayerZeroBridge.sol`, and `USX.sol` do not perform sufficient validation of the arguments provided. This could result in a denial of service attack if the contract initialization can be front-run.

Moreover, depending on the order of initialization, this may also pose a significant risk to cross-chain functionality.

Remediation

Validate the parameters of the `initialize` function and protect the functions to be only called by a specific caller.

src/bridging/layer_zero/LayerZeroBridge.sol SOLIDITY

```
function initialize(address _lzEndpoint, address _usx) public initializer
    ↪ {
        /// @dev No constructor, so initialize Ownable explicitly.
        require(msg.sender == address(0xdeadbeef}, "Invalid caller");
    ↪ //replace the address
        require(_lzEndpoint != address(0) && _usx != address(0), "Invalid
    ↪ Parameter");
        __Ownable_init();
        __NonBlockingLzApp_init_unchained(_lzEndpoint);
        usx = _usx;
}
```

src/bridging/wormhole/WormholeBridge.sol SOLIDITY

```
function initialize(address _wormholeCoreBridge, address _usx) public
    ↪ initializer {
        /// @dev No constructor, so initialize Ownable explicitly.
        require(msg.sender == address(0xdeadbeef}, "Invalid caller");
    ↪ //replace the address
        require(_wormholeCoreBridge != address(0) && _usx != address(0),
    ↪ "Invalid Parameter");
        __Ownable_init();
        wormholeCoreBridge = IWormhole(_wormholeCoreBridge);
        usx = _usx;
    }
```

src/token/USX.sol SOLIDITY

```solidity
function initialize() public initializer {
    /// @dev No constructor, so initialize Ownable explicitly.
    require(msg.sender == address(0xdeadbeef}, "Invalid caller"); //replace the address
    __Ownable_init();
    __ERC20_init("USX", "USX");
}
```

Patch

Resolved in 777ba1d.

OS-USX-ADV-02 [low] | Locked Ether In Contracts

Description

`Treasury.sol` and `USX.sol` allow for the receipt of ether through their `receive` functions but do not contain functions to withdraw it. As a result, if ether is sent to the contracts, it will become locked. Therefore, a function is needed that can be used to withdraw ether from the contracts.

src/token/USX.sol SOLIDITY

```solidity
    receive() external payable {}
```

src/token/Treasury.sol SOLIDITY

```solidity
    receive() external payable {}
```

Remediation

Add a function to extract ether from these contracts which is admin gated, similar to `extractERC20`.

SOLIDITY

```solidity
function extractNative() public onlyOwner {
    payable(msg.sender).transfer(address(this).balance);
}
```

Patch

Resolved in 777ba1d.

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 antipatterns and could lead to security issues in the future.

ID	Description
OS-USX-SUG-00	`get_virtual_price` is susceptible to read-only reentrancy attacks.
OS-USX-SUG-01	Implement more rigorous input validation for multiple functions.
OS-USX-SUG-02	Several improvements to optimize gas usage.

OS-USX-SUG-00 | Mitigate Curve LP Token Price Manipulation

Description

`Treasury.sol` calculates the `mintAmount` and `redeemAmount` by fetching the price of the LP tokens using `get_virtual_price`. This price may be manipulated temporarily with the help of read-only reentrancy. Since `get_virtual_price` does not have reentrancy protection, manipulation of the price may occur if reentrancy is possible within the curve pool operations.

This has been the root cause of several recent exploits, and it may be useful to mitigate this behaviour explicitly as a defence-in-depth measure.

Additionally, it may be safer to abort processing if the LP token value decreases, as this represents a violation of a critical invariant. It may make more sense to revert instead of silently accepting the original value in such cases.

src/treasury/Treasury.sol SOLIDITY

```solidity
    // Don't allow LP token price to decrease
    if (lpTokenPrice < previousLpTokenPrice) {
        lpTokenPrice = previousLpTokenPrice;
    } else {
        previousLpTokenPrice = lpTokenPrice;
    }
```

Remediation

Before fetching the price in `__getMintAmount` and `__getLpTokenAmount`, call any function which is reentrancy guarded such as `remove_liquidity`.

src/treasury/Treasury.sol SOLIDITY

```solidity
function __getMintAmount(uint256 _lpTokenAmount) private returns (uint256
    ↪  mintAmount) {
    uint256[2] calldata amounts;
    ICurve3Pool(token).remove_liquidity(0, amounts);
    uint256 lpTokenPrice = ICurve3Pool(CURVE_3POOL).get_virtual_price();
```


src/treasury/Treasury.sol SOLIDITY

```
function __getLpTokenAmount(uint256 _amount) private returns (uint256
    ↪  lpTokenAmount) {
    uint256[2] calldata amounts;
    ICurve3Pool(token).remove_liquidity(0, amounts);
    uint256 lpTokenPrice = ICurve3Pool(CURVE_3POOL).get_virtual_price();
```

Also consider refactoring the LP token price calculations to

src/treasury/Treasury.sol SOLIDITY

```
    // Don't allow LP token price to decrease
    require(lpTokenPrice >= previousLpTokenPrice);
    previousLpTokenPrice = lpTokenPrice;
```

Patch

Resolved in dac44d6.

OS-USX-SUG-01 | Stricter Input Validation

Description

The following are recommendations to improve input validation for key functions:

1. In `UERC20.sol`, validate that the address inputs to the following functions are nonzero.
   - `transfer`
   - `transferFrom`
   - `approve`
2. In `WormholeBridge::setSendFees`, validate that the lengths of `_destChainIds` and `_fees` are equal.
3. In `WormholeBridge` and `LayerZeroBridge`, validate the length of `_toAddress`. Otherwise, this may potentially allow unsafe address parsing.

LayerZeroBridge.sol SOLIDITY

```solidity
    assembly {
        toAddress := mload(add(toAddressBytes, 20))
    }
```

4. In `Treasury`, ensure a non-zero amount in the following functions.
   - `mint`
   - `redeem`
   - `stakeCrv`

Remediation

Add the described checks as a defense-in-depth measure.

Patch

Resolved in 777ba1d and dac44d6.

OS-USX-SUG-02 | Potential Gas Optimizations

Description

The following are suggestions to improve gas utilization:

- Use `calldata` instead of `memory` for `setSendFees` function parameters.
- Using `private` instead of `public` for constants uses less gas.
- Using `!=0` instead of `>0` for integer comparisons.

Remediation

- Replace the `memory` with `calldata` for parameters in `setSendFees`

src/bridging/wormhole/WormholeBridge.sol SOLIDITY

```solidity
function setSendFees(uint16[] calldata _destChainIds, uint256[]
    ↪ calldata _fees) public onlyOwner {
```

- Avoid initializing a variable with its default value, especially in for loops.

src/token/bridging/OERC20.sol SOLIDITY

```solidity
for (uint256 i; i < _destChainIds.length; i++) {
```

src/bridging/wormhole/WormholeBridge.sol SOLIDITY

```solidity
for (uint256 i; i < _bridgeAddresses.length; i++) {
```

- Use the `private` modifier over `public` for constants that do not require public visibility.

src/treasury/Treasury.sol SOLIDITY

```solidity
// Constants: no SLOAD to save gas
address private constant BACKING_TOKEN =
    ↪ 0x6c3F90f043a72FA612cbac8115EE7e52BDe6E490; // 3CRV
address private constant CURVE_3POOL =
    ↪ 0xbEbc44782C7dB0a1A60Cb6fe97d0b483032FF1C7;
address private constant CRV =
    ↪ 0xD533a949740bb3306d119CC777fa900bA034cd52;
address private constant CVX =
    ↪ 0x4e3FBD56CD56c3e72c1403e103b45Db9da5B9D2B;
```

```solidity
address private constant BOOSTER =
    ↪ 0xF403C135812408BFbE8713b5A23a04b3D48AAE31;
address private constant CRV_DEPOSITOR =
    ↪ 0x8014595F2AB54cD7c604B00E9fb932176fDc86Ae;
address private constant CVX3CRV_BASE_REWARD_POOL =
    ↪ 0x689440f2Ff927E1f24c72F1087E1FAF471eCe1c8;
address private constant CVXCRV_BASE_REWARD_POOL =
    ↪ 0x3Fe65692bfCD0e6CF84cB1E7d24108E434A7587e;
address private constant CVX_REWARD_POOL =
    ↪ 0xCF50b810E57Ac33B91dCF525C6ddd9881B139332;
uint8 private constant PID_3POOL = 9;
```

src/bridging/layer_zero/LayerZeroBridge.sol SOLIDITY

```solidity
// Constants: no SLOAD
uint256 private constant NO_EXTRA_GAS = 0;
uint256 private constant FUNCTION_TYPE_SEND = 1;
```

- Rewrite the expressions with `!=0`.

src/treasury/Treasury.sol SOLIDITY

```solidity
require(balance != 0 && balance >= _amount, "Insufficient CVX
    ↪ balance.");
```

src/treasury/Treasury.sol SOLIDITY

```solidity
require(stakedAmount != 0 && stakedAmount >= _amount, "Amount
    ↪ exceeds staked balance.");
```

src/treasury/Treasury.sol SOLIDITY

```solidity
require(stakedAmount != 0 && stakedAmount >= _amount, "Amount
    ↪ exceeds staked balance.");
```

src/treasury/Treasury.sol SOLIDITY

```solidity
require(balance != 0 && balance >= _amount, "Insufficient 3CRV
    ↪ balance.");
```

Patch

Resolved in 777ba1d and dac44d6.

A | Vulnerability Rating Scale

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

Critical	Vulnerabilities that immediately lead to 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 could lead to 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 could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	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 implementation of the program 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 sum, 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 get a comprehensive understanding of 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.