Connext xTokens Audit

Overview

Executive Summary

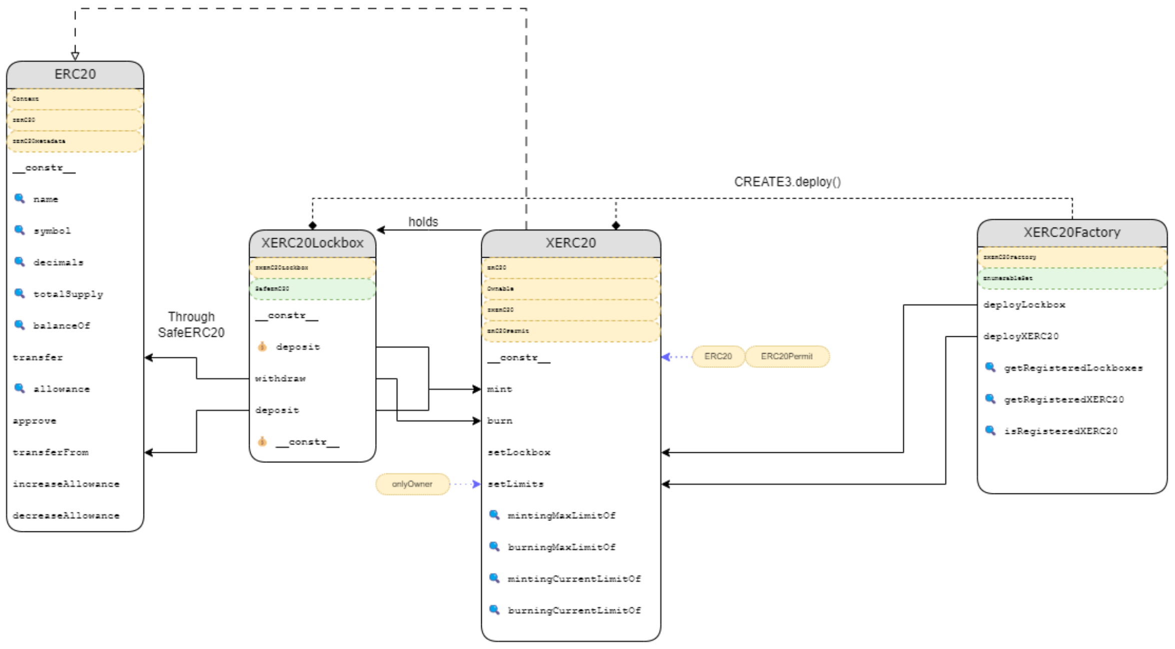
This report presents the results of our engagement with **Connext and Wonderland** to review their **xTokens** smart contracts system, a reference implementation of EIP-7281, implementing a common interface to keep liquidity concentrated across bridges.

The review was conducted by the Creed paladins, **Shayan Eskandari** and **Dominik Muhs** over the course of one week.

Scope

Repository	https://github.com/defi-wonderland/xTokens/tree/52d1e5c2e8671a05f8020ab61f5e204250d945d5/solidity/contracts
Revision	`52d1e5c2e8671a05f8020ab61f5e204250d945d5`

Architecture



Findings

[low] Immediate Settings Changes

When the `setLimits` function is called by the contract's owner, they can immediately change the minting and burning limits associated with a particular address.

https://github.com/defi-wonderland/xTokens/blob/52d1e5c2e8671a05f8020ab61f5e204250d945d5/solidity/contracts/XERC20.sol#L89-L93

```
function setLimits(address _bridge, uint256 _mintingLimit, uint256 _burningLimit) external onlyOwner {
    _changeMinterLimit(_mintingLimit, _bridge);
    _changeBurnerLimit(_burningLimit, _bridge);
    emit BridgeLimitsSet(_mintingLimit, _burningLimit, _bridge);
  }
```

Such immediate changes in contract settings can lead to unintended side effects. A concern is the potential interference with in-flight transactions, which might get reverted. To illustrate, consider an address initiating a `deposit` action on the `XERC20` contract. The contract owner modifies the limit to a lower value just before the transaction completes. This change can cause the user's transaction to fail, producing an `IXERC20_NotHighEnoughLimits` error. The same principle applies to the `withdraw` function.

Another risk this functionality introduces is that malicious owners might exploit these immediate changes to strategically sandwich user transactions with settings changes. This can be weaponized to single out and censor individual users' transactions.

This finding has been assigned a low severity rating since owners and bridges are assumed to be trustworthy parties, and they don't have an incentive to misbehave.

Recommendation

Consider introducing a delay or a time-lock mechanism for such critical changes. This would give users a notice period and minimize the risk of abruptly affecting transactions.

Alternatively, documentation should reference the potential issue, and developers integrating with the contracts should be briefed on how to avoid the issue of immediate changes on their side, e.g., by implementing an internal one-block delay whenever a setting is changed.

[low] Misleading Event Order

Due to an unprotected reentrancy vector, the sequence of emitted events on the `withdraw` function can be manipulated:

https://github.com/defi-wonderland/xTokens/blob/52d1e5c2e8671a05f8020ab61f5e204250d945d5/solidity/contracts/XERC20Lockbox.sol#L77

```
function withdraw(uint256 _amount) external {
    XERC20.burn(msg.sender, _amount);

    if (IS_NATIVE) {
      (bool _success,) = payable(msg.sender).call{value: _amount}('');
      if (!_success) revert IXERC20Lockbox_WithdrawFailed();
    } else {
      ERC20.safeTransfer(msg.sender, _amount);
    }

    emit Withdraw(msg.sender, _amount);
  }
```

When first a deposit is made, and in the same transaction, right after the `withdraw` function is used, an attacker can reenter through the external call in the `withdraw` function into the `deposit` function. This results in an unexpected sequence of emitted events: *Deposit, Deposit, Withdraw*.

This sequence can be misleading and cause confusion or incorrect handling by off-chain infrastructure that relies on monitoring and interpreting these events. The more intuitive and expected sequence of events should be *Deposit, Withdraw, Deposit*.

Recommendation

Implement a mechanism to prevent cross-function reentrancy in the contract. This can be achieved using a reentrancy guard. If a reentrancy guard presents too much overhead, consider emitting the event at the beginning of the `withdraw` function and revisiting the design to ensure that the sequence of events is emitted more intuitively.

Alternatively, the system's documentation should reference this issue and provide clear guidelines on consuming Lockbox events in the correct order.

[low] Missing Result Validation

The `XERC20Factory` contract dynamically deploys new XERC20-Lockbox pairs and tracks previous deterministic deployments in its registry state. However, upon deploying a new token and lockbox contract, the call adding the address to the internal state is not validated:

https://github.com/defi-wonderland/xTokens/blob/52d1e5c2e8671a05f8020ab61f5e204250d945d5/solidity/contracts/XERC20Factory.sol#L161

```
EnumerableSet.add(_xerc20RegistryArray, _xerc20);
```

The `EnumerableSet.add` function aims to add a new item to the set, and if the item does not already exist, it returns `True`. However, if the item is already present in the set, the function won't revert but will return `False`. This behavior might be overlooked as the return value of the `add` function isn't being checked in the code.

This issue has been marked as low severity since the `CREATE3.deploy` call ahead of the relevant line will most likely revert in this case since the target address already contains code and most likely a non-zero nonce.

Recommendation

Implement a check on the return value of the `EnumerableSet.add` function. If the function returns `False`, the contract should revert.

```
bool added = EnumerableSet.add(_xerc20RegistryArray, _xerc20);
if !added {
  revert IXERC20Factory_AlreadyRegistered();
}
```

Despite the low probability of this check ever being triggered, we recommend it nonetheless to enforce a defense-in-depth approach in the code base.

[info] Ambiguous Function `deposit` in `XERC20Lockbox`

In the `XERC20Lockbox` contract, the function `deposit` has potential ambiguity. The distinction between native and non-native tokens should be named explicitly. Once the `IS_NATIVE` value is set on deployment, there might be opportunities to simplify the deposit function.

Recommendation

Consider adopting clearer naming conventions for the native and non-native tokens to avoid confusion. Evaluate the possibility of simplifying the `deposit` function, especially after the `IS_NATIVE` value is determined at deployment.

This issue is informational, and while no immediate solution was identified, we also found no scenarios where the current implementation might lead to problems. It is worth noting that some VSCode extensions and potentially other tools do not handle overloaded functions correctly.

[info] Confusing Parameter Order

In the implementation, there's inconsistency in the parameter order. For instance, functions `_useMinterLimits` and `_useBurnerLimits` have the parameters `_change` and `_bridge`, respectively. Meanwhile, other functions such as `_mint` and `_mintWithCaller` start with target addresses followed by the amount.

Recommendation

To enhance developer experience and minimize potential errors, we recommend aligning the parameter order to be consistent with the ERC20 standard interface. Specifically, consider rearranging parameters so that target addresses precede the amount, ensuring consistent parameter ordering across functions.

[info] Duplicate Factory View Code

In the `XERC20Factory` contract, duplicate code exists between the functions `getRegisteredLockboxes` and `getRegisteredXERC20`. Both functions contain similar logic for validating the input `_start` and `_amount` parameters and for populating the result array.

https://github.com/defi-wonderland/xTokens/blob/52d1e5c2e8671a05f8020ab61f5e204250d945d5/solidity/contracts/XERC20Factory.sol#L81-L121

```
function getRegisteredLockboxes(uint256 _start, uint256 _amount) public view returns (address[] memory _lockboxes) {
    uint256 _length = EnumerableSet.length(_lockboxRegistryArray);  // @audit (info) duplicate code
    if (_amount > _length - _start) {
      _amount = _length - _start;
    }

    _lockboxes = new address[](_amount);
    uint256 _index;
    while (_index < _amount) {
      _lockboxes[_index] = EnumerableSet.at(_lockboxRegistryArray, _start + _index);

      unchecked {
        ++_index;
      }
    }
  }

  function getRegisteredXERC20(uint256 _start, uint256 _amount) public view returns (address[] memory _xerc20s) {
    uint256 _length = EnumerableSet.length(_xerc20RegistryArray);  // @audit (info) duplicate code
    if (_amount > _length - _start) {
      _amount = _length - _start;
    }

    _xerc20s = new address[](_amount);
    uint256 _index;
    while (_index < _amount) {
      _xerc20s[_index] = EnumerableSet.at(_xerc20RegistryArray, _start + _index);  // @audit-ok index strictly less than length

      unchecked {
        ++_index;
      }
    }
  }
```

Recommendation

Refactor the shared logic between `getRegisteredLockboxes` and `getRegisteredXERC20` into a common internal function to reduce code duplication, leading to a cleaner, more maintainable, and more concise codebase.

By having a shared utility function, potential changes or fixes will only need to be applied in one place, minimizing the risk of discrepancies between the functions in the future.

[info] Redundant Usage of `payable`

In the `XERC20Factory` contract's `_deployLockbox()` method, the `payable` keyword is used to store the address of the deployed lockbox in the `address _lockbox` variable.

https://github.com/defi-wonderland/xTokens/blob/52d1e5c2e8671a05f8020ab61f5e204250d945d5/solidity/contracts/XERC20Factory.sol#L179

```
    _lockbox = payable(CREATE3.deploy(_salt, _bytecode, 0));
```

Recommendation

Reevaluate the use of the `payable` keyword for storing the lockbox address. Depending on the outcome, ensure the corresponding lockbox contract address in the `XERC20` contract is also marked as `payable` for consistency. Otherwise, consider omitting the `payable` keyword for clarity and to avoid potential misinterpretations.

Disclaimer

The scope and objectives of this review are summarized in the Findings section. The matters raised in this report are only those identified during the review and are not necessarily a comprehensive statement of all weaknesses that exist or all actions that might be taken. The work was performed under limitations of time and scope that are potentially not relevant to the actions of a malicious attack. The review is based on the code in scope at a specific point in time, in an environment where both the code base system and the threat actors are dynamically evolving. It is therefore possible that vulnerabilities exist or will arise that were not identified during the review and there may or will have been events, developments, and changes in circumstances subsequent to its delivery.