SMART CONTRACT AUDIT REPORT

for

xBank Protocol

Prepared By: Patrick Lou

PeckShield
March 26, 2022

Document Properties

Client	xBank
Title	Smart Contract Audit Report
Target	xBank
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 26, 2022	Xuxian Jiang	Final Release
1.0-rc	March 18, 2022	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `xBank` protocol, we outline in the report our systematic approach to evaluate potential security issues in the implementation, expose possible semantic inconsistencies between code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About xBank

`xBank` is a decentralized non-custodial liquidity market protocol, written natively in `Cairo`. `xBank` manages deposits for the lenders and facilitates lending of the deposited asset for the borrowers while performing appropriate risk management to protect the lenders from risks of illiquidity and insolvency. The lenders earn passive interest income from the algorithmically derived interest rate while the borrowers take out a loan of the supplied assets in an overcollateralized and perpetual manner by paying a floating interest rate as determined by the supply and demand for the asset. The basic information of the `xBank` protocol is as follows:

Table 1.1: Basic Information of The `xBank` Protocol

Item	Description
Name	xBank
Type	DeFi/Lending
Language	Cairo
Audit Method	Whitebox
Latest Audit Report	March 26, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/xbank-lab/xbank-contract.git (ee2c1ba)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/xbank-lab/xbank-contract.git (54b7743)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the xBank implementation. During the first phase of our audit, we study the source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	2	■■
Informational	2	■■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 2 informational suggestions.

Table 2.1: Key xBank Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential Reentrancy Risk in xtoken Repayment	Time and State	Resolved
PVE-002	Low	Improved ERC20-Compliance Of xtoken	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Low	Improved Sanity Checks For System Parameters	Coding Practices	Resolved
PVE-005	Informational	Removal of Unused Imports	Coding Practices	Resolved
PVE-006	Informational	Proper log_add_reserves Events Upon Reserve Changes	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Potential Reentrancy Risk in xtoken Repayment

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `xtoken`
- Category: Time and State [7]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in smart contract development is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [12] exploit, and the recent `Uniswap/Lendf.Me` hack [11].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `xtoken` as an example, the `_repay_internal()` function (see the code snippet below) is provided to externally call a token contract to transfer assets (as repayment). However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 1180) start before effecting the update on internal states (lines 1185-1186), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
1150 func _repay_internal{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr}(
1151         _payer : felt, _borrower : felt, _repay_amount : Uint256) -> (
1152         actual_repay_amount : Uint256):
1153     alloc_locals
1154
```

```
    accrue_interest()

    let (local _total_borrows) = total_borrows.read()
    let (local _xcontroller) = xcontroller.read()
    let (local _self) = get_contract_address()
    let (local _borrow_index) = borrow_index.read()

    # validate based on xcontroller validation rules
    let (_is_repay_allowed) = IXcontroller.repay_allowed(
        _xcontroller, _self, _payer, _borrower, _repay_amount)
    assert _is_repay_allowed = 1

    # check repay amount > 0
    let (_is_repay_amount_gt_zero) = uint256_lt(Uint256(0, 0), _repay_amount)
    assert _is_repay_amount_gt_zero = 1

    # get the recent borrow balance (principal) + accrued interest of the _borrower
    let (_recent_borrow_balance) = _borrow_balance_stored_internal(_borrower)
    let (_is_borrow_balance_gt_zero) = uint256_lt(Uint256(0, 0), _recent_borrow_balance)
    assert _is_borrow_balance_gt_zero = 1

    # check _safe_repay_amount so payer won't overpay the debt
    let (_safe_repay_amount) = _get_safe_repay_amount(_repay_amount,
        _recent_borrow_balance)

    # transfer from payer to this contract
    let (_actual_repay_amount) = _do_transfer_in(_payer, _safe_repay_amount)

    # update total_borrows and account_borrows
    let (_updated_borrow_balance) = uint256_sub(_recent_borrow_balance,
        _actual_repay_amount)
    let (_updated_total_borrows) = uint256_sub(_total_borrows, _actual_repay_amount)
    total_borrows.write(_updated_total_borrows)
    account_borrows.write(
        _borrower, BorrowSnapshot(principal=_updated_borrow_balance, interest_index=
            _borrow_index))

    log_repay.emit(
        payer=_payer,
        borrower=_borrower,
        repay_amount=_actual_repay_amount,
        account_borrow_balance=_updated_borrow_balance,
        total_borrows=_updated_total_borrows,
        borrow_index=_borrow_index)

    return (actual_repay_amount=_actual_repay_amount)
end
```

Listing 3.1: `xtoken::_repay_internal()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`. And the adherence of the `checks-effects-interactions` best practice is strongly recommended.

From another perspective, the traditional mitigation in applying money-market-level reentrancy protection can be strengthened by elevating the reentrancy protection at the `xcontroller`-level. In addition, each individual function can be self-strengthened by following the `checks-effects-interactions` principle.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`. Also consider strengthening the reentrancy protection at the protocol-level instead of at the current money-market granularity.

Status The issue has been fixed by this commit: `19a6d19`.

3.2 Improved ERC20-Compliance Of xtoken

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `xtoken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

Each asset supported by the `xBank` is integrated through a so-called `xtoken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `xtoken`, users can earn interest through the `xtoken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `xtoken` as collateral.

In the following, we examine the ERC20 compliance of these `xtokens`. Note the ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there are minor ERC20 inconsistency or incompatibility issues found in the `xtoken` contract. Specifically, the current `transfer()` function does not properly emit the related `Transfer` event when the sender has the sufficient balance to spend. A similar issue is also present in the `transferFrom()` function that does not emit the related `Transfer` event.

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the `xtoken` implementation by emitting related `Transfer` events to better ensure its ERC20-compliance.

Status This issue can be fixed by upgrading the `erc20_base` contract from the latest `OpenZeppelin` implementation.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	—
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	—
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	—
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	—
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The `xBank` protocol has a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., `market` addition, oracle adjustment, and parameter setting). In the following, we show representative privileged operations in the protocol's core `xtoken` contract.

```
 @external
func set_xcontroller{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr}(
        _xcontroller : felt) -> (success : felt):
    ownable_only_owner()
    xcontroller.write(_xcontroller)
    log_set_xcontroller.emit(xcontroller=_xcontroller)
    return (1)
end

@external
func set_underlying{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr}(
        _underlying : felt) -> (success : felt):
    ownable_only_owner()
    underlying.write(_underlying)
```

```
    return (1)
end

@external
func set_reserve_factor{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*,
    range_check_ptr}(
        _factor : Uint256) -> (success : felt):
    alloc_locals
    accrue_interest()
    ownable_only_owner()
    # validate reserve factor range
    let (_valid_reserve_factor_check) = uint256_le(_factor, Uint256(MAX_RESERVE_FACTOR,
        0))
    assert _valid_reserve_factor_check = 1
    reserve_factor.write(_factor)

    log_set_reserve_factor.emit(factor=_factor)

    return (1)
end
```

Listing 3.2: `xtoken::set_xcontroller()/set_underlying()/set_reserve_factor()`

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Meanwhile, we point out that a compromised privileged account would allow the attacker to add a malicious underlying or change other settings, which directly undermines the assumption of the `xBank` protocol.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed. The team will have the owner transferred to a timelock contract or a multisig contract.

3.4 Improved Sanity Checks For System Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `xcontroller`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `xBank` protocol is no exception. Specifically, if we examine the `xtoken` contract, it has defined a number of protocol-wide risk parameters, such as `close_factor` and `liquidation_incentive`. In the following, we show the corresponding routines that allow for their changes.

```
@external
func set_close_factor{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr}(
        _xtoken : felt, _new_close_factor : Uint256) -> (bool : felt):
    alloc_locals
    ownable_only_owner()

    let (_xtoken_market) = markets.read(_xtoken)
    assert _xtoken_market.is_listed = 1

    # validate _new_close_factor
    _uint256_assert_in_range(
        _new_close_factor, Uint256(MIN_CLOSE_FACTOR, 0), Uint256(MAX_CLOSE_FACTOR, 0))

    # update _new_close_factor
    markets.write(
        _xtoken,
        Market(is_listed=1, collateral_factor=_xtoken_market.collateral_factor,
            close_factor=_new_close_factor))

    log_set_close_factor.emit(xtoken=_xtoken, is_listed=1, close_factor=
        _new_close_factor)
    return (bool=1)
end

@external
func set_liquidation_incentive{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*,
    range_check_ptr}(
        _new_liquidation_incentive : Uint256) -> (new_liquidation_incentive : Uint256):
    ownable_only_owner()

    let (_liquidation_incentive) = liquidation_incentive.read()
    liquidation_incentive.write(_liquidation_incentive)

```

```
    log_set_liquidation_incentive.emit(liquidation_incentive=_liquidation_incentive)
    return (new_liquidation_incentive=_new_liquidation_incentive)
end
```

Listing 3.3: xcontroller :: set_close_factor()/ set_liquidation_incentive ()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `liquidation_incentive` may yield unreasonably high amount in the `liquidate_calculate_xtoken_seizable()` calculation, hence incurring cost to liquidation or hurting the adoption of the protocol.

In the meantime, we notice that the `xtoken`'s `constructor()` configures a number of parameters. A specific one is the `_initial_supply`, which needs to be validated as zero. A non-zero `_initial_supply` may make the `xtoken` non-functional due to the reverted exchange rate calculation.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed by this commit: `19a6d19`.

3.5 Removal of Unused Imports

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `xmath_base`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The `xBank` protocol makes good use of a number of reference contracts, such as `ERC20`, `ownerable`, and `xmath`, to facilitate its code implementation and organization. For example, the `xtoken` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `xmath_base` implementation, there are a number of imports which are not used in the contract. These unused imports may be safely removed from importing.

```
%lang starknet

from starkware.cairo.common.cairo_builtins import HashBuiltin, SignatureBuiltin
```

```

from starkware.cairo.common.math import assert_not_equal, assert_not_zero
from starkware.cairo.common.math_cmp import is_le
from starkware.cairo.common.uint256 import (
    Uint256, uint256_lt, uint256_eq, uint256_sub, uint256_le, uint256_add, uint256_mul,
    uint256_unsigned_div_rem)
...
```

Listing 3.4: The `xmath_base` Library

Specifically, the unused `imports` include `assert_not_equal`, `assert_not_zero`, `is_le`, `uint256_lt`, `uint256_eq`, and `uint256_sub`. These imports become redundant and thus can be safely removed.

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `19a6d19`.

3.6 Proper log_add_reserves Events Upon Reserve Changes

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `xtoken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In the design of `DeFi` protocols, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `xtoken` contract as an example. This contract has public functions that may affect the accumulation of reserves. While examining the events that reflect the `reserve` changes, we notice there is a lack of emitting important events that reflect their changes. Specifically, when there is a liquidation being performed, the new `reserve` state may be reflected in `total_reserves`. However, no event is emitted to reflect its update (line 1311).

```
1276 func _seize_internal{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr}(
1277         _xtoken_seizer : felt, _liquidator : felt, _borrower : felt,
1278         _xtoken_seize_amount : Uint256) -> (actual_xtoken_seize_amount : Uint256):
1279     alloc_locals
```

```
    # validate based on xcontroller validation rules
    let (_self) = get_contract_address()
    let (_xcontroller) = xcontroller.read()
    IXcontroller.seize_allowed_check(
        _xcontroller, _self, _xtoken_seizer, _liquidator, _borrower,
            _xtoken_seize_amount)

    # assert _liquidator is not _borrower
    assert_not_equal(_liquidator, _borrower)

    let (_protocol_seize_share_factor) = protocol_seize_share_factor.read()
    let (_protocol_xtoken_seize_amount) = _uint256_mul_accurate(
        _xtoken_seize_amount, _protocol_seize_share_factor)
    let (_liquidator_xtoken_seize_amount) = uint256_sub(
        _xtoken_seize_amount, _protocol_xtoken_seize_amount)

    let (_exchange_rate) = _exchange_rate_stored_internal()
    let (_protocol_underlying_seize_amount) = _uint256_mul_accurate(
        _exchange_rate, _protocol_xtoken_seize_amount)

    let (_reserves_prior) = total_reserves.read()

    # ensure borrower will not get overseized
    let (_total_xtoken_seizing_from_borrower, _is_add_overflow) = uint256_add(
        _protocol_xtoken_seize_amount, _liquidator_xtoken_seize_amount)
    assert _is_add_overflow = 0
    assert _xtoken_seize_amount = _total_xtoken_seizing_from_borrower

    # platform will not own xtoken, accounting the underlying seize amount to the
        reserves and burn the share off
    let (_total_reserves_new, _is_add_overflow) = uint256_add(
        _reserves_prior, _protocol_underlying_seize_amount)
    assert _is_add_overflow = 0
    total_reserves.write(_total_reserves_new)
    ERC20_burn(_borrower, _protocol_xtoken_seize_amount)

    # force xtoken transfer to liquidator
    _transfer(_borrower, _liquidator, _liquidator_xtoken_seize_amount)
    # TODO: Emit transfer event
    # TODO: Emit seize event (?)
    return (actual_xtoken_seize_amount=_liquidator_xtoken_seize_amount)
end
```

Listing 3.5: `xtoken::_seize_internal`

Recommendation Properly emit respective events when the reserve state `total_reserves` is updated.

Status The issue has been fixed by this commit: `19a6d19`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `xBank` protocol, which is a decentralized non-custodial liquidity market protocol with a `Ethereum` Layer 2 (L2) scalability solution to power scalable self-custodial transactions (borrowing and lending) for DeFi users. The current code base is clearly organized and those identified issues are promptly confirmed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.

[11] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[12] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.