SMARTHUB INNOVATIONS

# BANKX AUDIT REPORT

## January 3, 2024

by

SmartHub Innovations
#SmartAudits
(Ridwan)

CONTENTS

## EXECUTIVE SUMMARY

**Project Name:** Bank X

**Overview:** BankX is A Dual-Token System Stablecoin: Stablecoin - BankX Silver Dollar (XSD).

**Method:** Manual Review, Functional Testing, Automated Testing etc.

**Scope of Audit:** The scope of this audit was to analyze Bank X codebase for quality, security, and correctness.

**Codebase:** https://github.com/BankXio/stablecoin

| | High | Medium | Low |
|---|---|---|---|
| Open Issues | 2 | 3 | 2 |
| Acknowledged Issues | 0 | 0 | 0 |
| Resolved & Closed | 0 | 0 | 0 |

## TYPES OF SEVERITIES

**High**
A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fifixed before moving to a live environment.

**Medium**
The issues marked as medium severity usually arise because of errors and defificiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fifixed.

**Low**
Low-level severity issues can cause minor impact and or are just warnings that can remain unfifixed for now. It would be better to fifix these issues at some point in the future.

**Informational**
These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

## TYPES OF ISSUES

**Open**
Security vulnerabilities identifified that must be resolved and are currently unresolved.

**Resolved**
These are the issues identifified in the initial audit and have been successfully fifixed.

**Acknowledged**
Vulnerabilities which have been acknowledged but are yet to be resolved.

## CHECKED VULNERABILITIES

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

## TECHNIQUES AND METHODS

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour
- mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

### Structural Analysis

In this step we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

### Static Analysis

Static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step a series of automated tools are used to test security of smart contracts.

### Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerability or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of automated analysis were manually verified.

**Gas Consumption**

In this step we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and possibilities of optimization of code to reduce gas consumption.

**Tools and Platforms used for Audit**

Remix IDE, Hardhat, Foundry, Solhint, Mythril, Slither, Solidity statistic analysis.

## ISSUES FOUND - MANUAL TESTING/CODE REVIEW

## HIGH SEVERITY ISSUES

### 1. Reentrancy Risk

### Contract - CollateralPool.sol

### Function - `collectRedemption`(334 - 370)

### Severity; High

```
function collectRedemption() external nonReentrant{
        require(!pid_controller.bucket3(), "Cannot withdraw in times of deficit");
        require(!redeem_paused, "Redeem Paused");
        require(((lastRedeemed[msg.sender]+(block_delay)) <= block.number) &&
        ((lastPriceCheck[msg.sender].lastpricecheck+(block_delay)) <= block.number) &&
        (lastPriceCheck[msg.sender].pricecheck), "Must wait for block_delay blocks before redeeming");
        uint BankXDollarAmount;
        uint CollateralDollarAmount;
        uint BankXAmount;
        uint CollateralAmount;

        // Use Checks-Effects-Interactions pattern
        if(redeemBankXBalances[msg.sender] > 0){
            BankXDollarAmount = redeemBankXBalances[msg.sender];
            BankXAmount = (BankXDollarAmount*1e6)/bankx_price;
            redeemBankXBalances[msg.sender] = 0;
            unclaimedPoolBankX = unclaimedPoolBankX-BankXDollarAmount;
            TransferHelper.safeTransfer(address(BankX), msg.sender, BankXAmount);
        }

        if(redeemCollateralBalances[msg.sender] > 0){
            CollateralDollarAmount = redeemCollateralBalances[msg.sender];
            CollateralAmount = (CollateralDollarAmount*1e6)/XSD.eth_usd_price();
            redeemCollateralBalances[msg.sender] = 0;
            unclaimedPoolCollateral = unclaimedPoolCollateral-CollateralDollarAmount;
            IWETH(WETH).withdraw(CollateralAmount); //try to unwrap eth in the redeem
            TransferHelper.safeTransferETH(msg.sender, CollateralAmount);
        }
        lastPriceCheck[msg.sender].pricecheck = false;
    }
```

### Description & Key Points of concern:

The identified reentrancy vulnerability occurs in the `collectRedemption` function, where the `TransferHelper.safeTransferETH` function is called after modifying state variables. Reentrancy vulnerabilities arise when external calls are made to other contracts before state changes are completed, allowing an attacker to potentially reenter the function and execute additional malicious code.

Here's a breakdown of the issue:

1. In the `collectRedemption` function, there is a sequence of operations where BankX and collateral are transferred to the user, and certain state variables are updated.

2. The `TransferHelper.safeTransferETH` function is called to transfer collateral to the user after the BankX transfer and state variable modifications.

3. Reentrancy can occur if the receiving address of `TransferHelper.safeTransferETH` is a malicious contract that calls back into the vulnerable contract before the state changes are completed.

4. During this reentrancy attack, the attacker's contract could execute additional logic and potentially manipulate the state of the vulnerable contract in unintended ways.

**Remediation**

To address this reentrancy vulnerability, the state changes should be completed before making any external calls. Specifically, the order of operations should be adjusted to ensure that critical state variables are updated only after the external transfers have been completed. This can be achieved by following the Checks-Effects-Interactions pattern, where external calls are the last steps in a function after state changes.

## 2A. Reentrancy Risk

## Contract - CollateralPool.sol

## Function - `mintFractionalXSD`(207 - 234)

## Severity; High

```
    (uint256 mint_amount, uint256 bankx_needed) = CollateralPoolLibrary.calcMintFractionalXSD(input_params);
    mint_amount = (mint_amount*31103477)/((xag_usd_price)); //grams of silver in calculated mint amount
    require(XSD_out_min <= mint_amount, "Slippage limit reached");
    require(bankx_needed <= bankx_amount, "Not enough BankX inputted");
    mintInterestCalc(mint_amount,msg.sender);
    bankx_minted_count = bankx_minted_count + bankx_needed;
    BankX.pool_burn_from(msg.sender, bankx_needed);
    IWETH(WETH).deposit{value: msg.value}();
    assert(IWETH(WETH).transfer(address(this), msg.value));
    collat_XSD = collat_XSD + mint_amount;
    lastPriceCheck[msg.sender].pricecheck = false;
    XSD.pool_mint(msg.sender, mint_amount);
}
```

## Description & Key Points of concern:

### i. External Calls Before State Changes:

The function makes several external calls (`mintInterestCalc`, `BankX.pool_burn_from`, `IWETH(WETH).deposit`, `IWETH(WETH).transfer`, and `XSD.pool_mint`) before modifying the contract state (`updating bankx_minted_count`, `collat_XSD`, and `lastPriceCheck`). This order can potentially lead to reentrancy attacks.

### ii. Reentrancy in External Calls:

If any of the external calls, especially `mintInterestCalc`, `BankX.pool_burn_from`, or `XSD.pool_mint`, triggers a fallback function that calls back into the contract, it could reenter the `mintFractionalXSD` function before the state changes are completed.

## Remediation

To mitigate this vulnerability, the common practice is to follow the **"checks-effects-interactions"** pattern. Perform state changes after external calls to prevent reentrancy attacks. Specifically, consider moving the state changes to occur after the external calls, reducing the risk of reentrancy.

```
// External calls
mintInterestCalc(mint_amount, msg.sender);
BankX.pool_burn_from(msg.sender, bankx_needed);
IWETH(WETH).deposit{value: msg.value}();
assert(IWETH(WETH).transfer(address(this), msg.value));
XSD.pool_mint(msg.sender, mint_amount);

// State changes
bankx_minted_count = bankx_minted_count + bankx_needed;
collat_XSD = collat_XSD + mint_amount;
lastPriceCheck[msg.sender].pricecheck = false;
```

By reordering the code in this way, the state changes occur after the external calls, reducing the risk of reentrancy attacks. Always exercise caution when interacting with external contracts and Ether transfers to avoid potential vulnerabilities.

## 2B. Reentrancy Risk

### Contract - CollateralPool.sol

### Function - `buyBackBankX` (376 -395)

```
(collateral_equivalent_d18) = (CollateralPoolLibrary.calcBuyBackBankX(input_params));

        require(COLLATERAL_out_min <= collateral_equivalent_d18, "Slippage limit reached");
        lastPriceCheck[msg.sender].pricecheck = false;
        // Give the sender their desired collateral and burn the BankX
        BankX.pool_burn_from(msg.sender, BankX_amount);
        TransferHelper.safeTransfer(address(WETH), address(this), collateral_equivalent_d18);
        IWETH(WETH).withdraw(collateral_equivalent_d18);
        TransferHelper.safeTransferETH(msg.sender, collateral_equivalent_d18);
    }
```

### Description & Key Points of concern:

**iii. External Calls Before State Changes:**
The function makes external calls (`BankX.pool_burn_from`, `TransferHelper.safeTransfer`, `IWETH(WETH).withdraw`, and `TransferHelper.safeTransferETH`) before modifying the contract state (updating `lastPriceCheck[msg.sender].pricecheck`). This order can potentially lead to reentrancy attacks.

**iv. Reentrancy in External Calls:**
If any of the external calls, especially `BankX.pool_burn_from` or any of the `TransferHelper` functions, triggers a fallback function that calls back into the contract, it could reenter the `buyBackBankX` function before the state changes are completed.

### Remediation

To mitigate this vulnerability, it is recommended to follow the **"checks-effects-interactions"** pattern. Perform state changes after external calls to prevent reentrancy attacks. Specifically, consider moving the state change (`lastPriceCheck[msg.sender].pricecheck = false;`) to occur after the external calls, reducing the risk of reentrancy.

```
// External calls
BankX.pool_burn_from(msg.sender, BankX_amount);
TransferHelper.safeTransfer(address(WETH), address(this), collateral_equivalent_d18);
IWETH(WETH).withdraw(collateral_equivalent_d18);
TransferHelper.safeTransferETH(msg.sender, collateral_equivalent_d18);

// State change
lastPriceCheck[msg.sender].pricecheck = false;
```

## 2C. Reentrancy Risk

**Contract - CollateralPool.sol**

**Function - `buyBackXSD` (397 -417)**

```
(collateral_equivalent_d18) = (CollateralPoolLibrary.calcBuyBackXSD(input_params));

require(collateral_out_min <= collateral_equivalent_d18, "Slippage limit reached");
lastPriceCheck[msg.sender].pricecheck = false;
XSD.pool_burn_from(msg.sender, XSD_amount);
TransferHelper.safeTransfer(address(WETH), address(this), collateral_equivalent_d18);
IWETH(WETH).withdraw(collateral_equivalent_d18);
TransferHelper.safeTransferETH(msg.sender, collateral_equivalent_d18);
```

**Description & Key Points of concern:**

**v. External Calls Before State Changes:**
The function makes external calls (`XSD.pool_burn_from`, `TransferHelper.safeTransfer`, `IWETH(WETH).withdraw`, and `TransferHelper.safeTransferETH`) before modifying the contract state (updating `lastPriceCheck[msg.sender].pricecheck`). This order can potentially lead to reentrancy attacks.

**vi. Reentrancy in External Calls:**
If any of the external calls, especially `XSD.pool_burn_from` or any of the `TransferHelper` functions, triggers a fallback function that calls back into the contract, it could reenter the `buyBackXSD` function before the state changes are completed.

**Remediation**

To mitigate this vulnerability, it is recommended to follow the **"checks-effects-interactions"** pattern. Perform state changes after external calls to prevent reentrancy attacks. Specifically, consider moving the state change (`lastPriceCheck[msg.sender].pricecheck = false;`) to occur after the external calls, reducing the risk of reentrancy.

```
// External calls
XSD.pool_burn_from(msg.sender, XSD_amount);
TransferHelper.safeTransfer(address(WETH), address(this), collateral_equivalent_d18);
IWETH(WETH).withdraw(collateral_equivalent_d18);
TransferHelper.safeTransferETH(msg.sender, collateral_equivalent_d18);

// State change
lastPriceCheck[msg.sender].pricecheck = false;
```

By reordering the code in this way, the state change occurs after the external calls, reducing the risk of reentrancy attacks.

## MEDIUM SEVERITY ISSUES

### 1. Divide Before Multiplication Precision

**Contract - CollateralPool.sol**

**Function - `redeemFractionalXSD` (270 - 311)**

**Severity:** Medium

```
function redeemFractionalXSD(uint256 XSD_amount, uint256 BankX_out_min, uint256 COLLATERAL_out_min) external nonReentrant {
    // ...

    // Vulnerability: Divide before multiplication
    uint256 bankx_dollar_value_d18 = XSD_amount - ((XSD_amount * global_collateral_ratio) / (1e6));
    bankx_dollar_value_d18 = (bankx_dollar_value_d18 * xag_usd_price) / 31103477;

    // Vulnerability: Divide before multiplication
    uint256 bankx_amount = (bankx_dollar_value_d18 * 1e6) / bankx_price;

    // Vulnerability: Divide before multiplication
    uint256 collateral_dollar_value = (XSD_amount * global_collateral_ratio) / (1e6);
    collateral_dollar_value = (collateral_dollar_value * xag_usd_price) / 31103477;

    // Vulnerability: Divide before multiplication
    uint256 collateral_amount = (collateral_dollar_value * 1e6) / XSD.eth_usd_price();

    // ...

    // Mitigation: Use parentheses to control order of operations
    uint256 current_accum_interest = (XSD_amount * mintMapping[msg.sender].accum_interest) / total_xsd_amount;

    // ...

    // Vulnerability: Divide before multiplication
    bankx_amount = bankx_amount + ((current_accum_interest * 1e6) / bankx_price);

    // ...
}
```

**Description:**
The code involves dividing values before multiplying them, potentially leading to unexpected results or precision loss.

Examples: `bankx_dollar_value_d18`, `bankx_amount`, `collateral_dollar_value`, `collateral_amount`, `current_accum_interest`, and `bankx_amount` (again)

**Potential Risks:**
The severity of these vulnerabilities depends on the specific values involved and the overall impact on the contract's intended behavior.

Precision loss or unexpected results from divide-before-multiply operations could lead to financial errors or unintended consequences.

**Remediation**

To mitigate these vulnerabilities, ensure that divisions are performed after relevant multiplications and use parentheses to explicitly control the order of operations. This helps maintain the intended logic and avoids unintended precision loss.

## 2.Unused Return

## Contract - XSDStablecoin.sol

## Function - `pool_price` (111 - 136)

## Severity: Medium

```
function pool_price(PriceChoice choice) internal view returns (uint256) {
    // Get the ETH / USD price first, and cut it down to 1e6 precision
    uint256 _eth_usd_price = (uint256(eth_usd_pricer.getLatestPrice())*PRICE_PRECISION)/(uint256(10) ** eth_usd_pricer_decimals);
    uint256 price_vs_eth = 0;
    uint256 reserve0;
    uint256 reserve1;

    if (choice == PriceChoice.XSD) {
        (reserve0, reserve1, ) = xsdEthPool.getReserves();
        if(reserve0 == 0 || reserve1 == 0){
            return 1;
        }
        price_vs_eth = reserve0/(reserve1); // How much XSD if you put in 1 WETH
    }
    else if (choice == PriceChoice.BankX) {
        (reserve0, reserve1, ) = bankxEthPool.getReserves();
        if(reserve0 == 0 || reserve1 == 0){
            return 1;
        }
        price_vs_eth = reserve0/(reserve1);  // How much BankX if you put in 1 WETH
    }
    else revert("INVALID PRICE CHOICE. Needs to be either 0 (XSD) or 1 (BankX)");

    // Will be in 1e6 format
    return _eth_usd_price/price_vs_eth;
}
```

## Description

The "Unused return" error in Solidity indicates that the result of an external call is obtained, but its value is not utilized or stored in a local or state variable. In the provided code, the function `eth_usd_pricer.getLatestPrice()` is called to get the latest ETH/USD price. However, the returned value is not assigned to any variable, and it is not used in further computations.

```
uint256 _eth_usd_price = (uint256(eth_usd_pricer.getLatestPrice()) * PRICE_PRECISION) / (uint256(10) ** eth_usd_pricer_decimals);
```

In this line, the result of `eth_usd_pricer.getLatestPrice()` is multiplied and divided to adjust its precision, but the `_eth_usd_price` variable, which holds this value, is not used in any subsequent calculations or returned from the function.

**Remediation**

To resolve this issue, you should ensure that the result of the external call is utilized in a meaningful way, whether by using it in calculations, storing it in a variable, or returning it from the function if it's part of the intended behavior. If the result is not needed, consider removing the unnecessary external call or modifying the code accordingly.

**3. Dubious Typecast**

**Contract - XSDStablecoin.sol**

**Function - `pool_price` (111 - 136)**

**Severity: Medium**

```
uint256 _eth_usd_price = (uint256(eth_usd_pricer.getLatestPrice()) * PRICE_PRECISION) / (uint256(10) ** eth_usd_pricer_decimals);
```

**Description**

In this line, there is a typecast of the result of `eth_usd_pricer.getLatestPrice()` to `uint256`. The term "dubious" suggests that the typecast might be considered uncertain or questionable.

Here's a breakdown of the expression:

1. **`eth_usd_pricer.getLatestPrice()`**: This function call is expected to return a value, possibly of a type that is not explicitly mentioned in the code snippet. The result might be an integer, fixed-point number, or another numeric type.

2. **`(uint256(eth_usd_pricer.getLatestPrice())`**: This part is casting the result of `eth_usd_pricer.getLatestPrice()` to `uint256`. This typecast might be necessary if the result type is not already `uint256`.

3. ***PRICE_PRECISION:** This multiplication is then performed with `PRICE_PRECISION`, possibly to scale the value to a desired precision.

4. **/ (uint256(10) ** eth_usd_pricer_decimals):** Finally, there is another typecast and division to adjust the precision based on `eth_usd_pricer_decimals`.

The "dubious" nature could arise from the uncertainty of the result type from `getLatestPrice()` and the potential risks associated with typecasting. It's crucial to ensure that the typecast is appropriate for the actual type of the result and that precision adjustments are performed accurately.

**Remediation**

To improve code clarity and reduce potential issues, you may want to check and ensure that the type of the result matches the expected type, and consider handling potential errors or edge cases more explicitly. Additionally, providing comments explaining the rationale behind the typecasting could be helpful for future developers working on the code.

## LOW SEVERITY ISSUES

### 1. Dangerous Usage of Timestamp

**Contract - CollateralPoolLibrary.sol**

**Function - `calcMintInterest` (44 - 67)**

**Severity: Low**

```
uint256 gram_price = (silver_price*(1e4))/(311035);
if(time == 0){
interest_rate = rate;
amount = XSD_amount;
time = block.timestamp;
}
```

**Description:**

The function `calcMintInterest` is designed to calculate and update interest-related values based on certain parameters. The key issue lies in the use of `block.timestamp` to track the passage of time. `block.timestamp` represents the current timestamp of the block being mined, and it is set by the miner who successfully mines the block.

Miners have some degree of control over the timestamp, within certain limits. While there are rules in place to prevent miners from setting timestamps too far in the future, they can manipulate timestamps to some extent. This manipulation can lead to inaccuracies in time-dependent calculations, potentially allowing malicious miners to exploit the system.

**Remediation:**

To address this issue, it's recommended to use alternative methods for tracking time or consider additional mechanisms to ensure the integrity of time-dependent calculations. For instance, using block numbers instead of timestamps or incorporating an external oracle for time-related information can enhance the security of such calculations.

**2. Missing Zero Check**

**Contract - CollateralPool.sol**

**Function - `constructor` (103- 126)**

**Severity: Low**

```
constructor(
    address _xsd_contract_address,
    address _bankx_contract_address,
    address _bankxweth_pool,
    address _xsdweth_pool,
    address _WETH,
    address _smartcontract_owner
) {
    require(
        (_xsd_contract_address != address(0))
        && (_bankx_contract_address != address(0))
        && (_WETH != address(0))
        && (_bankxweth_pool != address(0))
        && (_xsdweth_pool != address(0))
    , "Zero address detected");
    XSD = XSDStablecoin(_xsd_contract_address);
    BankX = BankXToken(_bankx_contract_address);
    xsd_contract_address = _xsd_contract_address;
    bankx_contract_address = _bankx_contract_address;
    xsdweth_pool = _xsdweth_pool;
    bankxweth_pool = _bankxweth_pool;
    WETH = _WETH;
    smartcontract_owner = _smartcontract_owner;
}
```

**Description:**

In Ethereum smart contracts, it's common practice to check whether the addresses passed as parameters are not set to the zero address (0x0000000000000000000000000000000000000000). This validation helps prevent certain types of errors and vulnerabilities.

The `require` statement is meant to ensure that none of the provided addresses are set to the zero address. However, it lacks a comprehensive check for each individual address. If any of the provided addresses are

set to the zero address, the constructor will revert with the error message "Zero address detected."

**Remediation:**

To improve this validation, it's recommended to explicitly check each address individually and provide more informative error messages about which specific address is set to the zero address.

This improvement provides more clarity on which address triggered the error in case of a zero address being detected. It's considered a best practice to perform individual checks for each address parameter in the constructor to enhance the security and maintainability of the smart contract.

## AUTOMATED TEST

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

## Closing Summary

In this report, we have considered the security of the BankX codebase. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low severity were found. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

## Disclaimer

**SmartAudits** smart contract audit is not a security warranty, investment advice, or an endorsement of the BankX Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the BankX Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

## ABOUT SMARTAUDITS

SmartAudits is a secure smart contracts audit platform designed by SmartHub Innovations Technologies.
We are a team of dedicated blockchain security experts and smart contract auditors
determined to ensure that Smart Contract-based Web3 projects can avail the latest and best
security solutions to operate in a trustworthy and risk-free ecosystem.