Security Audit Report for USDX

Date: November 1, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Synth-X
Target	USDX

Version History

Version	Date	Description
1.0	November 1, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-11-07 11:23:57+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

This audit focuses on the code repositories of the USDX [1] of Synth-X, The USDX stablecoin project enables users to exchange supported stablecoin assets for USDX or redeem supported assets from USDX. Additionally, USDX holders can stake their tokens to receive sUSDX and earn yield. Liquidity providers in USDX pools may also stake their LP tokens to earn shares. Staking will occur in a series of epochs, with eligibility for specific pools in each epoch. Reward computation and distribution are managed off-chain.

Please note that this audit covers only the following contracts:

- contracts/StakedUSDX.sol
- contracts/USDX.sol
- contracts/USDXLPStaking.sol
- contracts/USDXRedeem.sol
- contracts/USDXSales.sol
- contracts/USDXSilo.sol

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
USDX	Version 1	1bba197c23243bd7386d72d4082bf1ebc68d1e27
	Version 2	d77bb6d7637b4e92042a36661ab270d449d25496

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any war-

[1] https://github.com/Synth-X/usdx-contract

ranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist

* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology and Common Weakness Enumeration. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	High	Low
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.

- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **one** potential security issue. Besides, we have **three** recommendations and **eight** notes.

- Medium Risk: 1
- Recommendation: 3
- Note: 8

ID	Severity	Description	Category	Status
1	Medium	Inconsistency between implementation and documentation	DeFi Security	Confirmed
2	-	Unused contract inheritance	Recommendation	Confirmed
3	-	Improper comments and typos	Recommendation	Confirmed
4	-	Redundant code	Recommendation	Fixed
5	-	Potential donation attack risk	Note	-
6	-	Potential locked funds for the redeem process	Note	-
7	-	Potential centralization risk	Note	-
8	-	Potential arbitrage opportunities	Note	-
9	-	Insufficient slippage check for ERC-4626	Note	-
10	-	Potential fee calculation issue	Note	-
11	-	Unlock time delay	Note	-
12	-	Potential risk for blacklisted users	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Inconsistency between implementation and documentation

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description According to the documentation, `SOFT_RESTRICTED_ROLE` cannot deposit `USDX` to obtain `sUSDX` or withdraw `sUSDX` for `USDX` in the contract `StakedUSDX`. However, current implementation allows the role `SOFT_RESTRICTED_ROLE` to invoke the function `withdraw` to withdraw `sUSDX` for `USDX`, which is inconsistent compared with the documentation.

```
function _withdraw(
  address caller,
  address receiver,
  address _owner,
  uint256 assets,
  uint256 shares
) internal override nonReentrant notZero(assets) notZero(shares) {
```

```
  if (hasRole(FULL_RESTRICTED_STAKER_ROLE, caller) || hasRole(FULL_RESTRICTED_STAKER_ROLE,
      receiver)) {
    revert(Errors.OPERATION_NOT_ALLOWED);
  }
  super._withdraw(caller, receiver, _owner, assets, shares);
  _checkMinShares();
}
```

Listing 2.1: contracts/StakedUSDX.sol

Impact The `SOFT_RESTRICTED_ROLE` is able to exchange `sUSDX` for `USDX`, which is inconsistent with the description provided in the documentation.

Suggestion Revise the logic according to documentation.

Feedback from the project We have acknowledged this issue, however, it is by design.

2.2 Additional Recommendation

2.2.1 Unused contract inheritance

Status Confirmed

Introduced by `Version 1`

Description Although the contract `StakedUSDX` inherits from the contract `Pausable`, it does not utilize this functionality, as the contract implements its own pause feature using custom state variables `_pausedDeposit` and `_pausedWithdraw`.

```
contract StakedUSDX is Ownable2Step, AccessControl, Pausable, ReentrancyGuard, ERC20Permit,
    ERC4626, IStakedUSDX {
```

Listing 2.2: contracts/StakedUSDX.sol

Suggestion Remove unused contract inheritance.

Feedback from the Project We acknowledge this recommendation, however, it is by design.

2.2.2 Improper comments and typos

Status Confirmed

Introduced by `Version 1`

Description In the contract `StakedUSDX`, the function `buy()` contains a typo for the error code `Errors.CONFIG_SUPPORT_ASEETS`, as well as the parameter `underlingAmount` parameter in the struct of the contract `IUSDXRedeem`.

```
function buy(
  address _collateralAsset,
  uint256 _collateralAmount,
  address _custodianAddress
) external override whenNotPaused nonReentrant {
  require(_supportedAssets.contains(_collateralAsset), Errors.CONFIG_SUPPORT_ASEETS);
  require(_collateralAmount > 0, Errors.ZERO_AMOUNT_NOT_VALID);
```

```
require(_custodianAddress != address(0) && _custodianAddresses.contains(_custodianAddress),
    Errors.INVALID_ROUTE);
  _buy(_collateralAsset, _collateralAmount, _custodianAddress);
}
```

Listing 2.3: contracts/USDXSales.sol

```
struct Redemption {
  uint256 cooldownEnd;
  uint256 underlingAmount;
  uint256 usdxAmount;
}
```

Listing 2.4: contracts/IUSDXRedeem.sol

In the contract `USDXLPStaking`, there is ambiguous comments. The comment states that the owner cannot modify total staked or cooling down, but the data structure has two variables related to cooling down: `totalCoolingDown` (total amount of tokens in cool down) and `cooldown` (the length of the cooldown period), which makes the meaning of this comment ambiguous.

```
function updateStakeParameters(address token, uint8 epoch, uint248 stakeLimit, uint48 cooldown)
    external onlyOwner {
  require(cooldown <= _MAX_COOLDOWN_PERIOD, Errors.MAX_COOLDOWN_EXCEEDED);
  StakeParameters storage stakeParameters = stakeParametersByToken[token];
  // owner cannot modify total staked or cooling down
  stakeParameters.epoch = epoch;
  stakeParameters.stakeLimit = stakeLimit;
  stakeParameters.cooldown = cooldown;
  emit StakeParametersUpdated(token, epoch, stakeLimit, cooldown);
}
```

Listing 2.5: contracts/USDXLPStaking.sol

Suggestion Revise the typos and comments.

Feedback from the Project The improper comment is removed in `Version 2`. The others are not fixed.

2.2.3 Redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `USDXLPStaking` is unable to receive native tokens because it does not have the `receive()`, `fallback()`, or `payable` modifiers on any functions. Therefore, there is no need to implement a mechanism to rescue ETH tokens. Although there is a very small chance that native tokens could be sent to the contract via the `selfdestruct` instruction, this instruction is expected to be deprecated.

```
function rescueTokens(address token, address to, uint256 amount) external onlyOwner nonReentrant
    checkAmount(amount) {
  require(to != address(0), Errors.ZERO_ADDRESS_NOT_VALID);
  // contract should never hold ETH
```

```
  if (token == _ETH_ADDRESS) {
    (bool success, ) = to.call{value: amount}('');
    if (!success) revert(Errors.TRANSFER_FAILED);
  } else {
    IERC20(token).safeTransfer(to, amount);
    _checkInvariant(token);
  }
  emit TokensRescued(token, to, amount);
}
```

Listing 2.6: contracts/USDXLPStaking.sol

Suggestion Remove this redundant code.

2.3 Note

2.3.1 Potential donation attack risk

Introduced by `Version 1`

Description Though the protocol has checked the minimum liquidity after operations to prevent donation attack, there is a possibility for donation attacks in privileged functions.

Specifically, in the function `redistributeLockedAmount`, when the `to` address is set to 0, the shares of blacklisted users are burned as rewards. However, during this process, there is no validation for `_checkMinShares()`, which could potentially expose the contract to a donation attack.

```
function redistributeLockedAmount(address from, address to) external onlyOwner {
  require(hasRole(FULL_RESTRICTED_STAKER_ROLE, from) && !hasRole(FULL_RESTRICTED_STAKER_ROLE, to), Errors.OPERATION_NOT_ALLOWED);

  uint256 amountToDistribute = balanceOf(from);
  uint256 usdxToVest = previewRedeem(amountToDistribute);
  _burn(from, amountToDistribute);
  // to address of address(0) enables burning
  if (to == address(0)) {
    _updateVestingAmount(usdxToVest);
  } else {
    _mint(to, amountToDistribute);
  }

  emit LockedAmountRedistributed(from, to, amountToDistribute);
}
```

Listing 2.7: contracts/StakedUSDX.sol

2.3.2 Potential locked funds for the redeem process

Introduced by `Version 1`

Description In the contract `USDXRedeem`, users must follow a two-step process to withdraw funds. The first step involves calling the function `redeem`, which redeem funds to a cooldown period. The second step requires calling the function `claim` to withdraw the funds after the cooldown period. However, during the redeem process, the contract does not verify whether the current asset token balance is sufficient. This may result in users not able to withdraw staked funds.

2.3.3 Potential centralization risk

Introduced by `Version 1`

Description The protocol includes several privileged functions that can manipulate the funds in the protocol. If the `owner`'s private key is lost or maliciously exploited, it could potentially cause significant losses to users.

2.3.4 Potential arbitrage opportunities

Introduced by `Version 1`

Description The contracts `USDXSales` and `USDXRedeem` conduct USDX providing and redeeming for supported assets in a fixed 1:1 ratio. Besides, in the contract `USDXRedeem`, the function `redeem` allows users to arbitrarily specify the type of token to redeem. When there is an arbitrage opportunity for a particular token, it may impact the liquidity of the protocol.

For example, when the value of USDC drops to 0.9 USD, users may exchange their USDC for USDX, as the exchange rate in the contract is always 1:1. They could then use the USDX to redeem DAI or USDC, and by repeating this process, the entire protocol would be left with liquidity of tokens with dropped value (USDC in the provided example).

2.3.5 Insufficient slippage check for ERC-4626

Introduced by `Version 1`

Description The contract `StakedUSDX` implements ERC-4626, which allows EOAs to call `deposit()` and `mint()`. However, due to the lack of slippage control, the shares and asset outputs from these functions may be less than expected for EOAs.

2.3.6 Potential fee calculation issue

Introduced by `Version 1`

Description In the contracts `USDXSales` and `USDXRedeem`, the calculation of fees may be subject to precision loss. It leads to two consequences:

```
function _buy(address _collateralAsset, uint256 _collateralAmount, address _custodianAddress)
    internal {
  uint256 fee;
  if (feeRate > 0) {
      fee = (_collateralAmount * feeRate) / FEE_RATE_FACTOR;
  }
```

Listing 2.8: contracts/USDXSales.sol

1. When the `_collateralAmount` is small enough, the calculated fee can be zero.
2. When there are assets of different decimals and users purchase the same amount of USDX, the fee may be zero for one asset and non-zero for another.

In general, there are cases where the calculated fee is zero.

2.3.7 Unlock time delay

Introduced by `Version 1`

Description In the contract `StakedUSDX`, for the function `cooldownAssets`, if a user repeatedly unstakes without withdrawing, the unlock time is continuously pushed back.

```
function cooldownAssets(uint256 assets) external override whenNotWithdrawPaused ensureCooldownOn
    returns (uint256) {
  require(assets <= maxWithdraw(_msgSender()), Errors.EXCESSIVE_WITHDRAW_AMOUNT);

  uint256 shares = previewWithdraw(assets);

  cooldowns[_msgSender()].cooldownEnd = uint104(block.timestamp) + cooldownDuration;
  cooldowns[_msgSender()].underlyingAmount += assets;
```

Listing 2.9: contracts/StakedUSDX.sol

2.3.8 Potential risk for blacklisted users

Introduced by `Version 1`

Description The document states that blacklisted users are not allowed to withdraw sUSDX to obtain USDX. However, in the contract `StakedUSDX`, the function `unstake()` does not check to verify whether the user is blacklisted. This could lead to a situation where a user initiates a freezing request while still on the whitelist, but is blacklisted during the unfreezing process. In such a case, the user could still withdraw the `USDX` that was frozen prior to being blacklisted.

```
function unstake(address receiver) external override whenNotWithdrawPaused {
  UserCooldown storage userCooldown = cooldowns[msg.sender];
  uint256 assets = userCooldown.underlyingAmount;
  require(block.timestamp >= userCooldown.cooldownEnd, Errors.INVALID_COOLDOWN);

  userCooldown.cooldownEnd = 0;
  userCooldown.underlyingAmount = 0;

  SILO.withdraw(receiver, assets);
}
```

Listing 2.10: contracts/StakedUSDX.sol

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS