SALUS SECURITY

DEC 2024

CODE SECURITY ASSESSMENT

USDX

Overview

Project Summary

- Name: USDX
- Platform: EVM-compatible chains
- Language: Solidity
- Repository:
    - https://github.com/Synth-X/usdx-contract
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	USDX
Version	v4
Type	Solidity
Dates	Dec 04 2024
Logs	Nov 29 2024; Nov 30 2024; Dec 03 2024; Dec 04 2024

Vulnerability Summary

Total High-Severity issues	0
Total Medium-Severity issues	2
Total Low-Severity issues	1
Total informational issues	1
Total	4

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 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 with 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.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Centralization risk	Medium	Centralization	Acknowledged
2	User's cooling period may be reset	Medium	Business Logic	Acknowledged
3	The mintWithPermit() Is Prone to Reverting	Low	Business Logic	Acknowledged
4	Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom()	Informational	Business Logic	Acknowledged

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Centralization risk

Severity: Medium	Category: Centralization
Target: - contracts/StakedUSDX.sol	

Description

There is an `owner` privileged account in the `StakedUSDX` contract, and the `owner` can call the `rescueTokens()` function.. If `owner`'s private key is compromised, an attacker can withdraw all assets from the contract.

If the privileged accounts are plain EOA accounts, this can be worrisome and pose a risk to the other users.

Recommendation

We recommend transferring privileged accounts to multi-sig accounts with timelock governors for enhanced security. This ensures that no single person has full control over the accounts and that any changes must be authorized by multiple parties.

Status

This issue has been acknowledged by the team.

2. User's cooling period may be reset

Severity: Medium	Category: Business Logic
Target: - contracts/StakedUSDX.sol	

Description

contracts/StakedUSDX.sol:L212-213

```
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;

    _withdraw(_msgSender(), address(SILO), _msgSender(), assets, shares);

    return shares;
}
```

Users utilize the cooldownAssets() function to unlock funds. However, the function does not consider funds already in the cooling period when updating cooldownEnd. Each call resets the cooldown expiration time, potentially causing the same funds to have their cooling period reset.

For instance, the following scenarios may lead to funds incorrectly entering a new cooling period:

- When previously locked funds are about to unlock, but the user calls cooldownAssets() to unlock additional funds, causing the cooldown period for the old funds to be reset.
- When previously unlocked funds have not yet been unstaked, and the user initiates a new unlock, resulting in the already unlocked funds being forced back into a new cooling period.

Recommendation

Consider optimising the logic of cooldown.

Status

This issue has been acknowledged by the team.

3. The mintWithPermit() Is Prone to Reverting

Severity: Low	Category: Business Logic
Target: - contracts/StakedUSDX.sol	

Description

contracts/StakedUSDX.sol:L155-L170

```
function mintWithPermit(
    uint256 shares,
    address receiver,
    uint256 _deadline,
    uint8 _permitV,
    bytes32 _permitR,
    bytes32 _permitS
) public whenNotDepositPaused returns (uint256) {
    require(shares <= maxMint(receiver), 'ERC4626: mint more than max');

    uint256 assets = previewMint(shares);
    IERC20Permit(asset()).safePermit(_msgSender(), address(this), assets, _deadline,
_permitV, _permitR, _permitS);
    _deposit(_msgSender(), receiver, assets, shares);

    return assets;
}
```

The mintWithPermit() function uses previewMint to calculate the amount of assets required for minting and then executes ERC20Permit based on this value. However, the result of previewMint is highly susceptible to changes in the vault's balance, such as linear reward distribution or user operations, which can unexpectedly cause the Permit to revert.

Recommendation

Consider adding an assetAmount parameter to the function. Use this value for the permit operation and validate that assetAmount is greater than or equal to the result returned by previewMint().

Status

This issue has been acknowledged by the team.

2.3 Informational Findings

4. Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom()

Severity: Informational	Category: Business Logic
Target: - contracts/USDXSilo.sol	

Description

Tokens not compliant with the ERC20 specification could return false from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

```
Callers MUST handle false from returns (bool success). Callers MUST NOT assume that
false is never returned!
```

Recommendation

Consider using the SafeERC20 library implementation from OpenZeppelin and call safeTransfer or safeTransferFrom when transferring ERC20 token.

Status

This issue has been acknowledged by the team.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit d77bb6d:

File	SHA-1 hash
Migrations.sol	cd4b7d54827ac9daf013104be444f6c8e218fc65
StakedUSDX.sol	cd2dc55dc3a989eac465028cc83356a1af4f9953
USDX.sol	23c59b70a95dcf56a1e467ae0d25447dc0a16328
USDXLPStaking.sol	762fadf25dddf4f7ab6b9967871330a870985a93
USDXRedeem.sol	7f6a70457b050cdbf10cb09a887d15b8eb841aa9
USDXSales.sol	8af135b37d0754db95d34d7f92efec6e1502b370
USDXSilo.sol	825e716f3fd6e42e47a48b4eacae1f9f23fc7210

And we audited the commit a1bd3ac that introduced new features to the usdx-contract repository.

File	SHA-1 hash
USDXSales.sol	7609baacb57537139efe3b0f5fd8eff28cc5bda8

And we audited the commit 15f446f that introduced new features to the usdx-contract repository.

File	SHA-1 hash
Timelock.sol	d9ca7f24c51fe8a3d4a41d9ba1e6eef2c72941f1

Appendix 2 - About Project

This protocol is centered around the USDX stablecoin, allowing users to participate in the ecosystem through staking and trading while earning rewards and achieving flexible asset growth.