Zellic

Prepared for
Haichen Shen
Scroll

Prepared by
Weipeng Lai
Chongyu Lv
Zellic

October 2, 2025

USX
Smart Contract Security Assessment

Contents

About Zellic 4

1. Overview 4

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction 6

2.1. About USX 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings 10

3.1. Incorrect value for storage-layout constant 11

3.2. Lack of zero-amount check in StakedUSX's deposit and withdrawal flows 13

3.3. The `notifyRewards` function distributes previous undistributed rewards when no stakers exist 15

3.4. Governance lacks a setter for `withdrawalPeriod` 17

3.5. The `claimWithdraw` implementation differs from specification 19

3.6. The `transferUSDCForWithdrawal` function relies on stale USX accounting instead of live USDC balance 20

3.7. Missing validation for identical facet addresses in `replaceFacet` 22

4. Discussion **23**

4.1. Facet-change events log only the first selector 24

4.2. Storage not aligned with ERC-7201 namespaced layout 24

4.3. Minimize diamond surface to avoid selector shadowing 25

4.4. AssetManager must hold exact balance before replacement 26

4.5. Lack of slippage protection in StakedUSX 27

5. Threat Model **27**

5.1. AssetManager.sol 28

5.2. StakedUSX.sol 28

5.3. USX.sol 31

6. Assessment Results **32**

6.1. Disclaimer 33

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Scroll from September 25th to September 29th, 2025. During this engagement, Zellic reviewed USX's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the yield-distribution mechanism resistant to exploitation and manipulation?
- Are there any vulnerabilities that could compromise the asset's peg stability?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

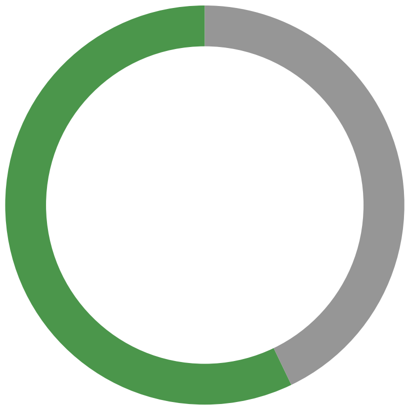
1.4. Results

During our assessment on the scoped USX contracts, we discovered seven findings. No critical issues were found. Four findings were of low impact and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Scroll in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	4
⬜ Informational	3



2. Introduction

2.1. About USX

Scroll contributed the following description of USX:

> Protocol USX is a fully collateralized neodollar that combines TradFi and DeFi to deliver ~12% APY, powered by previously inaccessible market-neutral hedge fund returns.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding’s impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

USX Contracts

Type	Solidity
Platform	EVM-compatible

Target	usx-contracts
Repository	https://github.com/scroll-tech/usx-contracts ↗
Version	`a249fa0fc4a21d943e6e53cb656b2f9d42c2b715`

Programs	`facets/RewardDistributorFacet.sol`
	`facets/AssetManagerAllocatorFacet.sol`
	`TreasuryStorage.sol`
	`TreasuryDiamond.sol`
	`StakedUSX.sol`
	`USX.sol`
	`asset-manager/AssetManager.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.2 person-weeks. The assessment was conducted by two consultants over the course of three calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Weipeng Lai
Engineer
weipeng.lai@zellic.io ↗

Chongyu Lv
Engineer
chongyu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 25, 2025	Kick-off call
September 25, 2025	Start of primary review period
September 29, 2025	End of primary review period

3. Detailed Findings

3.1. Incorrect value for storage-layout constant

Target	USX, StakedUSX		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The USX contract uses the ERC-7201 namespaced storage layout, but the slot constant is computed incorrectly:

```
// keccak256(abi.encode(uint256(keccak256("usx.main")) - 1)) &
    ~bytes32(uint256(0xff));
bytes32 private constant USX_STORAGE_LOCATION
    = 0x0c53c51c00000000000000000000000000000000000000000000000000000000;
```

Evaluating `keccak256(abi.encode(uint256(keccak256("usx.main")) - 1)) & ~bytes32(uint256(0xff))` yields `0xc9db443a76878c18b8727ca7977c3e648e5a60974201d1ee927d7e63744b5500`. The assigned value is incorrect.

The same issue occurs in StakedUSX, where `SUSX_STORAGE_LOCATION` is also incorrect:

```
// keccak256(abi.encode(uint256(keccak256("susx.main")) - 1)) &
    ~bytes32(uint256(0xff));
bytes32 private constant SUSX_STORAGE_LOCATION
    = 0x0c53c51c00000000000000000000000000000000000000000000000000000000;
```

Using the formula, the correct value is `0x7ef495ffa61cc9596b858592e81bad4189b8a35b6b875460d576397f44d3c900`.

Impact

Tools that rely on ERC-7201 storage namespaces resolve incorrect slots and fail to read state from these contracts.

Recommendations

We recommend using the correctly derived storage constants:

```
bytes32 private constant USX_STORAGE_LOCATION =
0x0c53c51c00000000000000000000000000000000000000000000000000000000;
bytes32 private constant USX_STORAGE_LOCATION =
0xc9db443a76878c18b8727ca7977c3e648e5a60974201d1ee927d7e63744b5500;

bytes32 private constant SUSX_STORAGE_LOCATION =
0x0c53c51c00000000000000000000000000000000000000000000000000000000;
bytes32 private constant SUSX_STORAGE_LOCATION =
0x7ef495ffa61cc9596b858592e81bad4189b8a35b6b875460d576397f44d3c900;
```

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit d8587d2c ↗.

3.2. Lack of zero-amount check in StakedUSX's deposit and withdrawal flows

Target	StakedUSX		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `_deposit` function in StakedUSX transfers `assets` USX from the caller and mints `shares` sUSX to the receiver. Because the function permits zero values for either parameter, it introduces the following edge cases:

- If `assets` equals zero while `shares` is nonzero, the caller mints sUSX without supplying USX.
- If `assets` is nonzero while `shares` equals zero, the caller transfers USX without receiving sUSX.
- If both `assets` and `shares` are zero, the call performs no meaningful action yet emits a `Deposit` event, cluttering logs.

The `_withdraw` function in StakedUSX burns `shares` from the owner and submits a withdrawal request for `assets`. It suffers from the same missing validation:

- If `assets` equals zero while `shares` is nonzero, the contract burns sUSX without queuing a USX withdrawal.
- If `assets` is nonzero while `shares` equals zero, the contract queues a withdrawal without burning sUSX.
- If both `assets` and `shares` are zero, the call does nothing substantive but still increments `withdrawalCounter` and records a zero-value `WithdrawalRequest`, enabling spam.

Impact

An attacker can front-run the first deposit by donating sufficient USX to the contract, causing the initial mint to round down to zero shares. This results in the first depositor receiving no shares despite transferring assets to the contract.

Attackers can call `withdraw` or `redeem` with zero amounts to generate unbounded numbers of zero-value withdrawal requests, potentially causing integration issues.

Recommendations

We recommend disallowing either `assets` or `shares` from being zero during deposits:

```
function _deposit(address caller, address receiver, uint256 assets,
    uint256 shares) internal nonReentrant override {
    if (assets == 0 || shares == 0) revert ZeroAmount();
    // [...]
}
```

Likewise, we recommend enforcing the same guard during withdrawals:

```
function _withdraw(address caller, address receiver, address owner,
    uint256 assets, uint256 shares)
    internal
    nonReentrant
    override
{
    if (assets == 0 || shares == 0) revert ZeroAmount();
    // [...]
}
```

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit 24548735 ↗.

3.3. The `notifyRewards` function distributes previous undistributed rewards when no stakers exist

Target	StakedUSX		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

After the last staker redeems their shares and before any new user stakes, the total supply of sUSX drops to zero. However, undistributed pending rewards may remain in the contract. A `notifyRewards` call in this scenario distributes the pending rewards immediately instead of adding them to `queued`.

```
function _increaseRewards(
    RewardData memory _data,
    uint256 _periodLength,
    uint256 _amount
) internal view {
    _amount = _amount + _data.queued;
    _data.queued = 0;

    // no supply, all rewards are queued
    if (totalSupply() == 0) {
        _data.rate = 0;
        _data.lastUpdate = uint40(block.timestamp);
        _data.finishAt = uint40(block.timestamp + _periodLength);
        _data.queued = uint96(_amount);
        return;
    }

    // [...]
}
```

The `notifyRewards` function calls `_increaseRewards`, which ignores pending rewards when `totalSupply()` is zero. This causes those rewards to be distributed immediately.

Impact

When `notifyRewards` executes with no stakers present, it immediately distributes previously undistributed pending rewards. An attacker could sandwich this `notifyRewards` call — depositing before and redeeming after — then call `claimWithdraw` once the withdrawal window elapses to capture the instantly released rewards.

Recommendations

We recommend that `notifyRewards` adds any remaining rewards from the current period to `queued` along with the new rewards when `totalSupply` is zero:

```
if (totalSupply() == 0) {
   if (block.timestamp < _data.finishAt) {
       _amount += uint256(_data.rate) * (_data.finishAt - block.timestamp);
   }
    _data.rate = 0;
    _data.lastUpdate = uint40(block.timestamp);
    _data.finishAt = uint40(block.timestamp + _periodLength);
    _data.queued = uint96(_amount);
    return;
}
```

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit `a4cf34cf` ↗.

3.4. Governance lacks a setter for `withdrawalPeriod`

Target	StakedUSX		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `withdrawalPeriod` state variable is set to 15 days when the StakedUSX contract is deployed. After deployment, governance cannot modify this value because no setter function exists for `withdrawalPeriod`.

The contract includes a `setMinWithdrawalPeriod` function that allows governance to set `minWithdrawalPeriod`, which appears intended as the minimum threshold for `withdrawalPeriod` modifications:

```
function setMinWithdrawalPeriod(uint256 _minWithdrawalPeriod)
    public onlyGovernance {
    if (_minWithdrawalPeriod < MIN_WITHDRAWAL_PERIOD)
    revert InvalidMinWithdrawalPeriod();
    SUSXStorage storage $ = _getStorage();
    uint256 oldPeriod = $.minWithdrawalPeriod;
    $.minWithdrawalPeriod = _minWithdrawalPeriod;
    emit WithdrawalPeriodSet(oldPeriod, _minWithdrawalPeriod);
}
```

The presence of `minWithdrawalPeriod` and its setter suggests that a corresponding `setWithdrawalPeriod` function was intended but not implemented.

Impact

Governance cannot adjust `withdrawalPeriod` after deployment, preventing the protocol from adapting to changing operational requirements or market conditions.

Recommendations

We recommend implementing a governance-controlled `setWithdrawalPeriod` function that enforces the constraint `withdrawalPeriod >= minWithdrawalPeriod`.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit `bb2ad497` ↗.

3.5. The `claimWithdraw` implementation differs from specification

Target	StakedUSX		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The documentation comment for `claimWithdraw` in StakedUSX asserts that withdrawals can be claimed only after the epoch in which they were requested has concluded. The current implementation does not enforce this epoch check.

```
/// @notice Finishes a withdrawal, claiming a specified withdrawal claim
/// @dev Allowed after withdrawalPeriod AND epoch the user made withdrawal on
    is finished, after Gross Profits has been counted
///     Portion is sent to the Governance Warchest (withdrawalFee applied here)
/// @param withdrawalId The id of the withdrawal to claim
function claimWithdraw(uint256 withdrawalId) public nonReentrant {
    // [...]
}
```

Impact

If the documentation reflects the intended behavior, users can bypass the epoch restriction and claim withdrawals early.

However, if the comment is outdated, it introduces ambiguity about the function's actual constraints.

Recommendations

Add epoch validation to the function if the comment accurately describes the intended behavior. Otherwise, update the comment to reflect the current implementation.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit `0b8d34ec` ↗.

3.6. The `transferUSDCForWithdrawal` function relies on stale USX accounting instead of live USDC balance

Target	AssetManagerAllocatorFacet		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The function `transferUSDCForWithdrawal` in AssetManagerAllocatorFacet calculates the shortfall by subtracting `$.USX.totalMatchedWithdrawalAmount()` from `$.USX.totalOutstandingWithdrawalAmount()`:

```
function transferUSDCForWithdrawal() external onlyAllocator nonReentrant {
    TreasuryStorage.TreasuryStorageStruct storage $ = _getStorage();

    uint256 totalOutstandingWithdrawalAmount
    = $.USX.totalOutstandingWithdrawalAmount();
    uint256 totalMatchedWithdrawalAmount
    = $.USX.totalMatchedWithdrawalAmount();
    uint256 missingUSDCForWithdrawal = totalOutstandingWithdrawalAmount
    - totalMatchedWithdrawalAmount;

    if (missingUSDCForWithdrawal > 0) {
        $.USDC.safeTransfer(address($.USX), missingUSDCForWithdrawal);
    }
    emit USDCTransferredForWithdrawal(missingUSDCForWithdrawal);
}
```

However, the value `totalMatchedWithdrawalAmount` updates only when `_updateTotalMatchedWithdrawalAmount` executes within USX. This occurs during `deposit`, `requestUSDC`, or `claimUSDC` operations. When these functions have not executed since the last USDC balance increase, `transferUSDCForWithdrawal` reads stale accounting data and underestimates the actual USDC balance in USX.

Impact

Using stale data causes the contract to transfer more USDC than necessary to USX for withdrawal fulfillment.

Recommendations

We recommend adding a public function in the USX contract that calls `_updateTotalMatchedWithdrawalAmount(false)` internally. This function should be callable only by the Treasury. Call this function before reading `totalMatchedWithdrawalAmount` in `transferUSDCForWithdrawal` to ensure the data is current.

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit `e7afce3b` ↗.

3.7. Missing validation for identical facet addresses in `replaceFacet`

Target	TreasuryDiamond		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `replaceFacet` function in TreasuryDiamond assumes the incoming facet differs from the outgoing facet. When `_oldFacet == _newFacet`, the final `delete` statement clears the selector list, leaving the facet with no registered functions in `facetFunctionSelectors`.

```
function replaceFacet(address _oldFacet, address _newFacet)
    external onlyGovernance {
    if (_newFacet == address(0)) revert ZeroAddress();

    bytes4[] memory selectors = facetFunctionSelectors[_oldFacet];

    for (uint256 i = 0; i < selectors.length; i++) {
        bytes4 selector = selectors[i];
        facets[selector] = _newFacet;
    }

    // Update facet arrays
    for (uint256 i = 0; i < facetAddresses.length; i++) {
        if (facetAddresses[i] == _oldFacet) {
            facetAddresses[i] = _newFacet;
            break;
        }
    }

    // Transfer selectors to new facet
    facetFunctionSelectors[_newFacet] = facetFunctionSelectors[_oldFacet];
    delete facetFunctionSelectors[_oldFacet];

    emit FacetReplaced(selectors[0], _oldFacet, _newFacet);
}
```

Impact

If governance mistakenly supplies the same facet address for both parameters, `facetFunctionSelectors` for that facet becomes deleted.

Recommendations

We recommend adding an explicit check to prevent equal addresses:

```
function replaceFacet(address _oldFacet, address _newFacet)
    external onlyGovernance {
    if (_newFacet == address(0)) revert ZeroAddress();
    if (_oldFacet == _newFacet) revert InvalidFacet();
    // [...]
}
```

Remediation

This issue has been acknowledged by Scroll, and a fix was implemented in commit `5b7f9b9f`.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Facet-change events log only the first selector

The `FacetAdded`, `FacetRemoved`, and `FacetReplaced` events in TreasuryDiamond currently log only the first affected selector, limiting observability of facet modifications.

```
function addFacet(address _facet, bytes4[] calldata _selectors)
    external onlyGovernance {
    // [...]
    emit FacetAdded(_selectors[0], _facet);
}

function removeFacet(address _facet) external onlyGovernance {
    // [...]
    emit FacetRemoved(selectors[0]);
}

function replaceFacet(address _oldFacet, address _newFacet)
    external onlyGovernance {
    // [...]
    emit FacetReplaced(selectors[0], _oldFacet, _newFacet);
}
```

These events emit only `selectors[0]`, omitting information about additional selectors associated with the facet operation. This incomplete logging prevents off-chain systems from accurately tracking all selector-to-facet mappings through event monitoring alone.

We recommend modifying the events to emit the facet addresses along with all associated selectors.

4.2. Storage not aligned with ERC-7201 namespaced layout

TreasuryDiamond stores `facets`, `facetAddresses`, and `facetFunctionSelectors` in the contract's storage rather than in an ERC-7201 namespace.

```
contract TreasuryDiamond is Initializable, UUPSUpgradeable,
    ReentrancyGuardUpgradeable, TreasuryStorage {
```

```
    // [...]

    // Mapping from function selector to facet address
    mapping(bytes4 => address) public facets;

    // Array of all facet addresses
    address[] public facetAddresses;

    // Mapping from facet address to array of function selectors
    mapping(address => bytes4[]) public facetFunctionSelectors;

    // [...]
}
```

We recommend moving the facet-management state into a dedicated ERC-7201 namespace to ensure consistency, upgrade safety, and collision avoidance.

Moreover, AssetManager uses a storage slot constant that does not follow the ERC-7201 scheme.

```
// keccak256("asset-manager.main")
bytes32 private constant ASSET_MANAGER_STORAGE_LOCATION =
    0x80eb0cdd16cb622196d298bc0913d06d921aa48c024ad1954e8e8ef1bbf9387d;
```

We recommend updating the storage location to comply with ERC-7201:

```
// keccak256("asset-manager.main")
// keccak256(abi.encode(uint256(keccak256("asset-manager.main")) - 1)) & ~
    bytes32(uint256(0xff));
bytes32 private constant ASSET_MANAGER_STORAGE_LOCATION =
    0x80eb0cdd16cb622196d298bc0913d06d921aa48c024ad1954e8e8ef1bbf9387d;
    0xde282b4c51a1df38ec1c6abe7d3af4a304418f4234a686e0dde0c8e3f160a500;
```

This issue has been acknowledged by Scroll, and a partial fix was implemented in commit `fe49fae0` ↗. The storage location in AssetManager has now been updated to comply with ERC-7201.

4.3. Minimize diamond surface to avoid selector shadowing

Calls with selectors matching functions declared on TreasuryDiamond never reach the fallback function for facet dispatch and always execute the diamond's function. For example, `removeFacet(address)` has the selector `0x0340e905`. If a facet unintentionally exposes a function with the same selector, calls trigger the diamond's `removeFacet(address)` function instead of the facet's version.

Although we did not find such collisions in the current codebase, we recommend keeping TreasuryDiamond minimal and moving external functions to facets to prevent selector collisions during future upgrades.

4.4. AssetManager must hold exact balance before replacement

The `setAssetManager` function withdraws the entire `$.assetManagerUSDC` balance from the current asset manager before funding the replacement.

```
function setAssetManager(address _assetManager) external onlyGovernance {
    // [...]
    if (oldAssetManager != address(0)) {
        // [...]
        IAssetManager(oldAssetManager).withdraw($.assetManagerUSDC);
        // [...]
    }
    // [...]
}
```

The `withdraw` function in AssetManager can only transfer the USDC it currently holds. However, the `deposit` function immediately forwards all USDC to weighted accounts. Therefore, these accounts must return the assets before governance replaces the contract via `setAssetManager`.

```
function withdraw(uint256 _usdcAmount) external onlyTreasury {
    IERC20(USDC).safeTransfer(treasury, _usdcAmount);
}

function deposit(uint256 _usdcAmount) external onlyTreasury {
    // [...]
    for (uint256 i = 0; i < $.weights.length(); i++) {
        (address account, uint256 weight) = $.weights.at(i);
        uint256 amount = (balance * weight) / totalWeight;
        IERC20(USDC).safeTransfer(account, amount);
        // [...]
    }
}
```

The AssetManager contract must hold exactly `$.assetManagerUSDC` USDC before calling `setAssetManager`. If the balance is insufficient, the transaction reverts. If the contract holds excess USDC, `setAssetManager` withdraws only `$.assetManagerUSDC`, leaving the remainder locked in the contract unless governance upgrades the AssetManager contract.

We recommend ensuring the AssetManager contract holds exactly `$.assetManagerUSDC` USDC before calling `setAssetManager`.

4.5. Lack of slippage protection in StakedUSX

The ERC-4626 interaction functions in StakedUSX lack slippage-protection mechanisms. For example, users call `deposit()` expecting to receive a specific number of shares based on the current preview, but the actual amount received can differ due to share price between transaction submission and execution. Without slippage protection, users cannot specify a minimum acceptable amount of shares they are willing to receive, potentially resulting in unexpected outcomes.

We recommend implementing slippage protection by adding ERC-5143 ↗ functions.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. AssetManager.sol

Function: `deposit(uint256 _usdcAmount)`

This function is used to deposit USDC to the asset manager.

Inputs

- `_usdcAmount`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The amount of USDC to deposit.

Branches and code coverage

Intended branches

- ☑ Test that when there is no weight (`totalWeight == 0`), a deposit only pulls USDC from the treasury to the manager contract and does not distribute.
- ☑ Set the weight first, and then distribute it to all users in proportion after deposit.
- ☑ After removing a certain weight, depositing again should only distribute to the remaining accounts.

Negative behavior

- ☑ Revert if the caller is not the treasury.

5.2. StakedUSX.sol

Function: `_deposit(address caller, address receiver, uint256 assets, uint256 shares)`

This function is used to transfer assets (USX) from the caller and mint shares (sUSX) to the receiver.

Inputs

- `caller`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Users who deposit USX.
- `receiver`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Users who receive sUSX.
- `assets`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The amount of underlying assets to be deposited this time (USX).
- `shares`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The sUSX shares to be minted this time.

Branches and code coverage

Intended branches

- [x] Test user deposit of USX can successfully mint the correct amount of sUSX.

Negative behavior

- [x] Revert if the contract is paused by governance.
- [ ] Revert if `assets == 0`.
- [ ] Revert if `shares == 0`.

Function: `_withdraw(address caller, address receiver, address owner, uint256 assets, uint256 shares)`

This function is used to burn shares from the owner and submits a withdrawal request for assets.

Inputs

- `caller`
    - **Control**: Arbitrary.

    - **Constraints**: None.
    - **Impact**: The operator who initiated this redemption on behalf of the owner.
- `receiver`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The address that receives the underlying USX token during subsequent claims.
- `owner`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Accounts whose sUSX shares were deducted and burned.
- `assets`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The amount of underlying USX corresponding to this redemption.
- `shares`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The sUSX shares burned this time.

Branches and code coverage

Intended branches

- ☑ Test the logic of the creation of a withdrawal request and update the status when redeeming.
- ☑ Test that after maturity, part of the fee is transferred to `governanceWarchest`, the rest is sent to the user, the treasury assets are reduced, and the request is marked as received.
- ☑ Test `caller != owner` consumes owner → caller shares authorization, and the receiver can be different from the owner.

Negative behavior

- ☑ Revert if `claimWithdraw` is called before the `withdrawalPeriod` has expired.
- ☑ Revert if there is a second claim.
- ☐ Revert if `assets == 0`.
- ☐ Revert if `shares == 0`.

5.3. USX.sol

Function: `claimUSDC()`

This function allows partial claims if there is some USDC available for users' total claim.

Branches and code coverage

Intended branches

- ☑ Test the complete flow: `deposit` → `requestUSDC` (creates withdrawal request) → `claimUSDC`.
- ☑ Test multiple withdrawal requests.
- ☑ Test that withdrawal requests are properly cleaned up.
- ☑ Test a partial claim when the contract has insufficient USDC for the full request.
- ☑ Test multiple partial claims over time.
- ☑ Test a partial claim when the contract has exactly the requested amount.
- ☑ Test partial claims with multiple users.

Negative behavior

- ☑ Revert if `NoOutstandingWithdrawalRequests`.
- ☑ Test that claiming with zero contract balance reverts.

Function: `deposit(uint256 _amount)`

This function is used to deposit USDC to get USX.

Inputs

- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: The `_amount` cannot be 0.
  - **Impact**: The amount of USDC to deposit.

Branches and code coverage

Intended branches

- ☑ Test that if a user deposits 100e6 USDC, they can correctly receive 100e18 USX.
- ☑ Test that when a user deposits a large amount of USDC, they can still be minted the corresponding USX correctly.

- ☑ Test that when a user deposits 1 WEI, the deposit decimal scaling is calculated correctly.

Negative behavior

- ☑ Revert if the user is not whitelisted.
- ☑ Revert if the amount deposited by the user is 0.

Function: `requestUSDC(uint256 _USXredeemed)`

This function is used to redeem USX to get back USDC (automatically sends if available, otherwise creates withdrawal request).

Inputs

- `_USXredeemed`
    - **Control**: Arbitrary.
    - **Constraints**: The amount cannot be 0 and must be a multiple of `USDC_SCALAR`.
    - **Impact**: The amount of USX to redeem.

Branches and code coverage

Intended branches

- ☑ Test the complete flow: `deposit` → `requestUSDC` with automatic USDC transfer.
- ☑ Test multiple withdrawal requests with mixed automatic transfer and fallback behavior.
- ☑ Test that if the current USDC of the contract is insufficient, the outstanding request can be correctly recorded.
- ☑ Test that the requested USDC outstanding amount is tracked correctly.

Negative behavior

- ☑ Test the fallback behavior when USDC is not available on the contract.
- ☑ Request zero-amount withdrawal should revert with `InvalidUSXRedeemAmount`.
- ☑ Revert if `_USXredeemed` is not an integer multiple of `USDC_SCALAR`.
- ☑ Revert if the contract is paused.

6. Assessment Results

During our assessment on the scoped USX contracts, we discovered seven findings. No critical issues were found. Four findings were of low impact and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.