Zellic

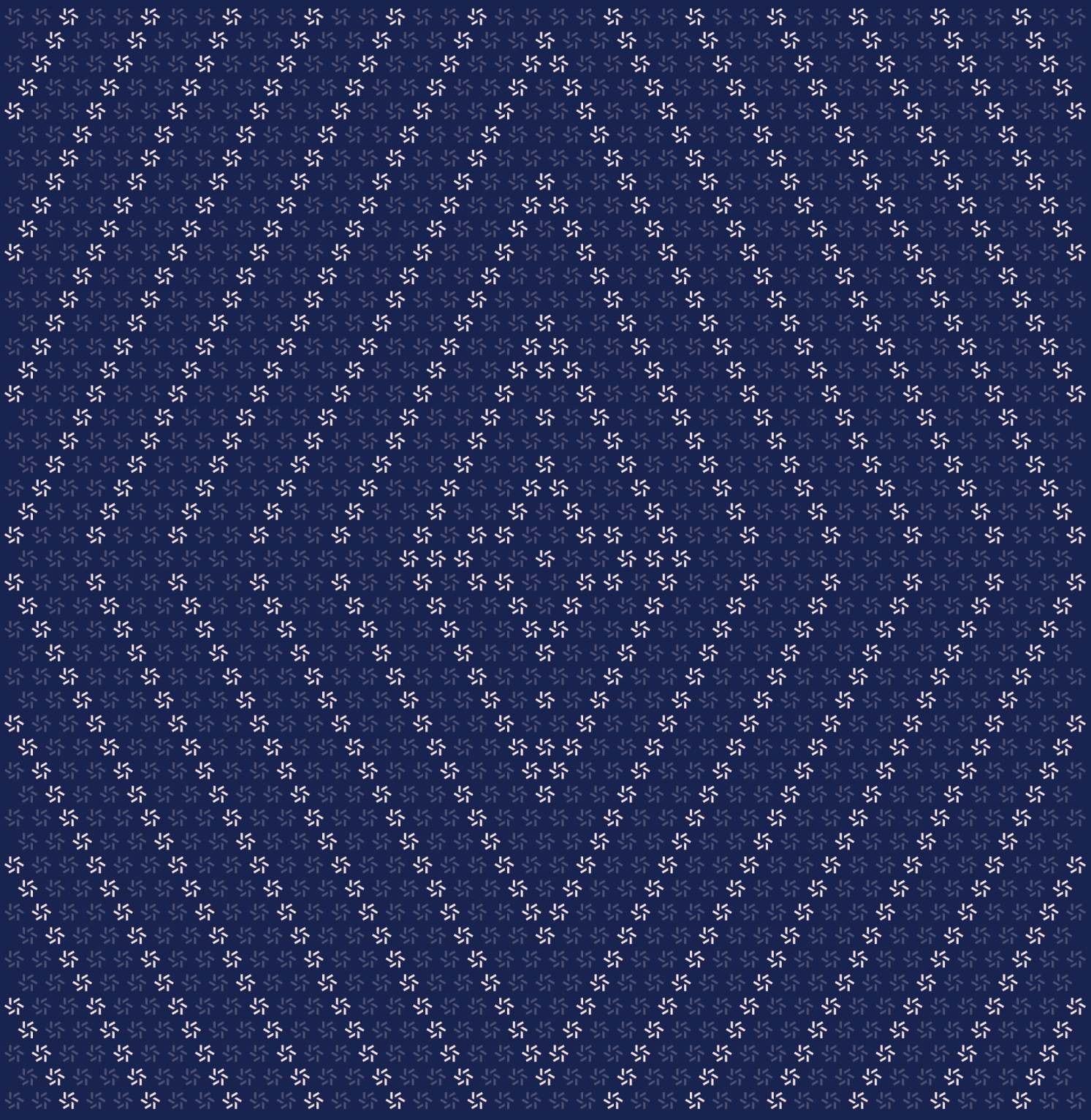
Prepared for
Coinshift
Coinshift

Prepared by
Juchang Lee
Chongyu Lv
Zellic

November 17, 2025

Multisafe USPC Contracts

Smart Contract Security Assessment



Contents

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 Coinshift from November 11th to November 14th, 2025. During this engagement, Zellic reviewed Multisafe USPC Contracts'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:

- Can allowlist/denylist/pausable controls be bypassed in token, vault, or hub transfer flows?
- Are fee and minimum amount calculations safe from rounding exploits and consistent across 6/36 decimals?
- Any reentrancy/external call patterns that can be abused during settlement?

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

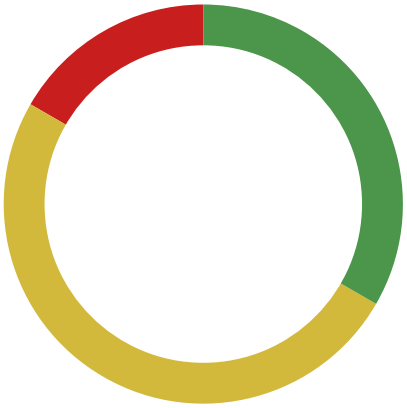
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Multisafe USPC Contracts contracts, we discovered six findings. One critical issue was found. Three were of medium impact and two were of low impact.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	0
Medium	3
Low	2
Informational	0



2. Introduction

2.1. About Multisafe USPC Contracts

Coinshift contributed the following description of Multisafe USPC Contracts:

> iUSPC / USPC is an institutional two-token framework with upgradeable ERC20/ERC4626 smart contracts and granular role-based controls:
>
> - iUSPC: a NAV-linked ERC20 security token with allowlist and pause controls, intended for qualified investors. Minting/redemption occurs via an on-chain hub using off-chain collateral settlement (e.g., USDC).
> - USPC: an ERC-4626 vault that wraps iUSPC 1:1 to enable DeFi composability. It adds withdrawal/transfer controls and donation-attack mitigation while delegating economic rights to iUSPC.
> - IUSPCHub: coordinates subscription/redemption using NAV prices from a Pricer contract with 36-decimals precision, supports fees, minimums, and whitelist-based "instant" flows for approved vaults.
> - Pricer: a controlled price feed storage with deviation checks and pausability used by the hub for settlement amounts.
>
> The system uses UUPS upgradeability and OpenZeppelin access control with distinct roles for upgrades, pausing, compliance, pricing, and operations.

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.

2.3. Scope

The engagement involved a review of the following targets:

Multisafe USPC Contracts Contracts

Type	Solidity
Platform	EVM-compatible
Target	uspc-contracts
Repository	https://github.com/Multisafe/uspc-contracts ↗
Version	`faf06f42d12560adff5940ccccf31d8460243a59`

Programs	`AdminTimelock.sol`
	`IUSPC.sol`
	`IUSPCHub.sol`
	`Pricer.sol`
	`USPC.sol`
	`base/AccessControlledPausableUpgradeable.sol`
	`base/AccessControlledUUPSUpgradeable.sol`
	`base/AllowList.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.1 person-weeks. The assessment was conducted by two consultants over the course of four calendar days.

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Juchang Lee
Engineer
lee@zellic.io ↗

Chongyu Lv
Engineer
chongyu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 11, 2025	Start of primary review period
November 14, 2025	End of primary review period

3. Detailed Findings

3.1. iUSPC Tokens are not burned during instant vault redemptions

Target	IUSPCHub		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `vaultRedemption` function in `IUSPCHub` allows whitelisted vaults to instantly redeem iUSPC for collateral. The function correctly transfers iUSPC from the vault to the Hub and transfers the corresponding USDC out. However, it fails to burn the iUSPC tokens received by the Hub. This contrasts with the standard `processRedemptions` function, which correctly burns the accumulated tokens.

Impact

This is a critical accounting error that breaks the fundamental backing of the system. When a vault redeems, collateral is paid out, but the liability remains in circulation, trapped within the `IUSPCHub`. The total supply of iUSPC becomes inflated relative to the assets under management, making the system fractionally reserved.

```
function testVaultRedemptionDoesNotBurnTokens() public {
    vm.startPrank(admin);
    hub.addVaultToWhitelist(vault);
    vm.stopPrank();

    // Setup vault with iUSPC
    usdc.mint(vault, USDC_AMOUNT);
    vm.startPrank(vault);
    usdc.approve(address(hub), USDC_AMOUNT);
    hub.vaultSubscription(USDC_AMOUNT);

    uint256 iuspcBalance = iuspc.balanceOf(vault);
    vm.stopPrank();

    uint256 totalSupplyBefore = iuspc.totalSupply();
    uint256 hubBalanceBefore = iuspc.balanceOf(address(hub));
    console2.log("hubBalanceBefore: ",hubBalanceBefore);

    // Vault redemption
    vm.startPrank(vault);
```

```
    iuspc.approve(address(hub), iuspcBalance);
    hub.vaultRedemption(iuspcBalance);
    vm.stopPrank();

    uint256 totalSupplyAfter = iuspc.totalSupply();
    uint256 hubBalanceAfter = iuspc.balanceOf(address(hub));

    // Verify iUSPC was transferred to hub but NOT burned
    assertEq(hubBalanceAfter, hubBalanceBefore + iuspcBalance, "iUSPC should
    be in hub");
    assertEq(totalSupplyAfter, totalSupplyBefore, "Total supply should NOT
    decrease (BUG: tokens not burned)");
    console2.log("hubBalanceAfter: ",hubBalanceAfter);
}
```

Recommendations

We recommend to modify the `vaultRedemption` function to burn the iUSPC tokens immediately.

Remediation

This issue has been acknowledged by Coinshift, and a fix was implemented in commit d14d8915 ↗.

3.2. Missing price staleness check in `IUSPCHub.vaultSubscription` and `IUSPCHub.vaultRedemption`

Target	IUSPCHub		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `IUSPCHub.vaultSubscription` and `IUSPCHub.vaultRedemption` functions provide instant processing capabilities for whitelisted Vaults, allowing users to directly retrieve prices using `pricer.getLatestPrice()`:

```
// Get current price for immediate processing
uint256 currentPrice = $.pricer.getLatestPrice();

// [...]
/**
     * @notice Gets the latest price of the asset
     * @return The latest price with 36 decimal precision
     */
    function getLatestPrice()
    external view override whenNotPaused returns (uint256) {
        PricerStorage storage $ = _getPricerStorage();
        require($.latestPriceId != 0, NoPricesSet());
        return $.prices[$.latestPriceId].price;
    }
```

The function calls `pricer.getLatestPrice()` to `vaultSubscription` and `vaultRedemption` only return the price value, not the timestamp. The code does not call `pricer.getLatestPriceInfo()` to retrieve the timestamp, nor does it perform the check that `block.timestamp - timestamp <= MAX_PRICE_AGE`.

Price staleness checks are correctly implemented in both the standard processes `IUSPCHub.processSubscriptions` and `IUSPCHub.processRedemptions`:

```
require(block.timestamp - timestamps[i] <= MAX_PRICE_AGE, PriceTooOld());
```

Impact

Consider the following scenario:

1. The price admin calls `Pricer.addCurrentPrice(price)` to update the NAV price, but due to various reasons (system failure, human error, etc.), the price is not updated for an extended period.
2. As time passes, the timestamp of the latest price exceeds `MAX_PRICE_AGE` (24 hours).
3. A regular user calls `requestSubscription` and waits for the administrator to process it. When the administrator calls `processSubscriptions`, the subscription is incorrectly rejected by `PriceTooOld` because the price has expired, preventing the user from completing the subscription.
4. Simultaneously, a Vault holding `VAULT_ROLE` and on the whitelist calls `vaultSubscription(amount)`, which successfully mints iUSPC tokens using the expired price.
5. The Vault can then call `vaultRedemption` to redeem USDC using the same expired price, completing arbitrage.

Recommendations

Consider adding `require(block.timestamp - timestamps[i] <= MAX_PRICE_AGE, PriceTooOld());` in `IUSPCHub.vaultSubscription` and `IUSPCHub.vaultRedemption`.

Remediation

This issue has been acknowledged by Coinshift, and a fix was implemented in commit `3e3b0fde` ↗.

3.3. Incorrect implementation of USPC._checkMinShares causes DOS.

Target	ContractNameHere		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In the USPC contract, the `_checkMinShares` function guarantees that after all deposits and withdrawals, if `_totalSupply` is not equal to 0 or is less than `MIN_SHARES`, then it will revert:

```
/**
     * @notice Internal deposit function with minimum shares check
     * @dev Overrides ERC4626 _deposit to add donation attack protection
     * @inheritdoc ERC4626Upgradeable
     */
    function _deposit(address _caller, address _receiver, uint256 _assets,
    uint256 _shares)
        internal
        override
        whenNotPaused
    {
        super._deposit(_caller, _receiver, _assets, _shares);
        _checkMinShares();
    }
// [...]
/**
     * @notice Internal withdraw function with minimum shares check
     * @dev Overrides ERC4626 _withdraw to add donation attack protection
     * @dev Throws if owner, caller, or receiver is not allowlisted
     * @inheritdoc ERC4626Upgradeable
     */
    function _withdraw(address _caller, address _receiver, address _owner,
    uint256 _assets, uint256 _shares)
        internal
        override
        whenNotPaused
    {
        require(isAllowlisted(_owner), OwnerNotAllowlisted());
        require(_owner == _caller || isAllowlisted(_caller),
    CallerNotAllowlisted());
```

```
        require(_owner == _receiver || _caller == _receiver ||
    isAllowlisted(_receiver), ReceiverNotAllowlisted());
        super._withdraw(_caller, _receiver, _owner, _assets, _shares);
        _checkMinShares();
    }
// [...]
/**
     * @notice Ensures a small non-zero amount of shares does not remain,
    exposing to donation attack
     * @dev Prevents attackers from donating small amounts to manipulate share
    price
     */
    function _checkMinShares() internal view {
        uint256 _totalSupply = totalSupply();
        require(_totalSupply == 0 || _totalSupply >= MIN_SHARES,
    MinSharesViolation()); // @audit: revert if `0 < _totalSupply < 10000e6`
    }
```

This introduces the possibility of a DOS attack.

Impact

Consider the following scenario:

1. User1 deposits 10000e6 USDC and receives 10000e6 shares.
2. The attacker deposits 1 wei.
3. User1 executes the `_withdraw` function to get 10000e6 USDC. At this point, `_totalSupply` is 1 wei, causing `require(_totalSupply == 0 || _totalSupply >= MIN_SHARES, MinSharesViolation());` to revert.

In this case, attackers can prevent users from withdrawing at extremely low cost.(1 wei)

Recommendations

Consider defending with a virtual offset ↗

- Use an offset between the "precision" of the representation of shares and assets. Said otherwise, we use more decimal places to represent the shares than the underlying token does to represent the assets.
- Include virtual shares and virtual assets in the exchange rate computation. These virtual assets enforce the conversion rate when the vault is empty.

Remediation

This issue has been acknowledged by Coinshift, and a fix was implemented in commit d14d8915 ↗.

3.4. Flawed Pause Logic

Target	Pricer, USPC		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The contract implementation contains faulty pause logic.

- The `_update` function in `USPC` does not have `whenNotPaused` modifier.
- Price reading functions (e.g., `getLatestPrice`) are protected by `whenNotPaused` modifier, but price management functions (`addPrice`, `updatePrice`) are not.
- The `processRedemptions` function applies only the global `whenNotPaused` modifier. However, the validation for the `redemptionPaused` flag has been omitted.

Impact

This incorrect logic renders the pause mechanism ineffective, resulting in a loss of control in emergency scenarios.

Recommendations

We recommend to add missing `whenNotPaused` modifier and `redemptionPaused` check.

Remediation

This issue has been acknowledged by Coinshift, and a fix was implemented in commit `d14d8915` ↗.

3.5. Missing Pricer Setter in IUSPCHub

Target	Pricer		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The pricer address is configured only at initialization, and the contract lacks a mechanism to update it subsequently.

Impact

If the existing pricer contract malfunctions or requires replacement, there is no mechanism to update it. This creates a risk of the protocol relying on incorrect price feeds.

Recommendations

We recommend to add a privileged function to allow updating the pricer contract address.

Remediation

This issue has been acknowledged by Coinshift, and a fix was implemented in commit `d14d8915` ↗.

3.6. Ineffective Fee Collection on Micro-Transactions

Target	ContractNameHere		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The protocol fails to account for precision loss inherent in integer division. Without a minimum fee enforcement, calculations for small input amounts resolve to zero. This creates a loophole where users can split their capital into dust amounts to avoid paying any fees during mint or redemption processes.

Impact

This oversight compromises the protocol's revenue stream, as fees can be systematically bypassed. Furthermore, it opens the door to DoS vectors where the network is congested with zero-fee dust transactions, disregarding the protocol's intended financial design.

Recommendations

We recommend to add fallback logic to enforce a minimum fee or minimum transaction size in minting or redemption.

Remediation

This issue has been acknowledged by Coinshift, and a fix was implemented in commit `d14d8915` ↗.

4. 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.

4.1. Module: IUSPCHUB.sol

Function: `processRedemptions(byte[32][] redemptionIds, uint256[] priceIds)`

This function is used to process redemptions of iUSPC tokens.

Inputs

- `redemptionIds`
  - **Control**: Full control.
  - **Constraints**: `redemptionIds.length` must be equal to `priceIds.length`, and each `redemptionIds` must exist in `redemptionIdToRedeemer`.
  - **Impact**: Array of redemption IDs.
- `priceIds`
  - **Control**: Full control.
  - **Constraints**: It must correspond to a valid oracle price, and `block.timestamp - timestamp <= MAX_PRICE_AGE`; the quantity must match `redemptionIds`.
  - **Impact**: Array of corresponding price IDs.

Branches and code coverage

Intended branches

- ☑ The authorized role successfully processed the redemption in bulk, and the user received USDC based on the net amount, with fees accrued.
- ☑ Price inquiry, amount calculation, and processing of funds and shares were successful.

Negative behavior

- ☑ Revert if `redemptionIds.length != priceIds.length`.
- ☑ Revert if caller is not `PRICE_ID_SETTER_ROLE`.
- ☑ Revert if the price timestamp expires.

Function: `processSubscriptions(byte[32][] depositIds, uint256[] priceIds)`

This function is used to process subscriptions with immediate processing.

Inputs

- `depositIds`
  - **Control**: Full control.
  - **Constraints**: `depositIds.length` must be equal to `priceIds.length`, and each `depositIds` must exist in `depositIdToDepositor`.
  - **Impact**: Array of deposit IDs.
- `priceIds`
  - **Control**: Full control.
  - **Constraints**: The difference between the current timestamp and the `timestamp` returned by `pricer.getPriceInfos` must be less than or equal to `MAX_PRICE_AGE`.
  - **Impact**: Array of corresponding price IDs.

Branches and code coverage

Intended branches

- ☑ When `PRICE_ID_SETTER_ROLE` calls this function, subscriptions are settled normally, and users receive the corresponding amount of iUSPC.
- ☑ After setting `mintFee`, the actual iUSPC received by the user is consistent with expectations.

Negative behavior

- ☑ Revert if a user is removed from `Allowlist` after initiating a subscription request.
- ☑ Revert if the difference between the current timestamp and the `timestamp` returned by `pricer.getPriceInfos` is greater than `MAX_PRICE_AGE`.
- ☐ Revert if `depositIds` is invalid or has already been processed.
- ☐ Revert if `depositIds.length` is not equal to `priceIds.length`.

Function: `requestRedemption(uint256 amount)`

This function is used to request redemption of iUSPC tokens.

Inputs

- `amount`
  - **Control**: Full control.
  - **Constraints**: The caller must pass the `isAllowlisted` check, the contract must be in the `whenNotPaused` state and `redemptionPaused` must be false, and the requested redemption amount must be greater than or equal to `minimumRedemptionAmount`.
  - **Impact**: Amount of iUSPC tokens to redeem.

Branches and code coverage

Intended branches

- ☑ Users can successfully initiate redemption after calling the function, and the `redemptionIdToRedeemer` mapping record is correct.
- ☑ When `allowlistEnabled` is true, users can initiate redemption after being added to `Allowlist`.

Negative behavior

- ☑ When `allowlistEnabled` is true, users cannot initiate redemption after being removed from `Allowlist`.
- ☑ When `RedemptionsPaused` is true, users cannot initiate redemption.
- ☑ Users cannot initiate redemption if the redemption amount is less than `minimumRedemptionAmount`.

Function: `requestSubscription(uint256 amount)`

This function is used to request a subscription (mint) of iUSPC tokens.

Inputs

- `amount`
  - **Control**: Full control.
  - **Constraints**: The caller must pass the `isAllowlisted` check, `subscriptionPaused` must be false, and the requested subscription amount must be greater than or equal to `minimumDepositAmount`.
  - **Impact**: Amount of collateral to deposit.

Branches and code coverage

Intended branches

- ☑ After the user calls the function, they can successfully initiate a subscription and generate a valid `depositId`.
- ☑ When `allowlistEnabled` is true, after adding the user to `Allowlist`, the user can initiate a subscription.

Negative behavior

- ☑ When `allowlistEnabled` is true and the user has not been added to `Allowlist`, the user cannot initiate a subscription.
- ☑ When `subscriptionPaused` is true, the user cannot initiate a subscription.
- ☑ When the subscription amount is too small, the user cannot initiate a subscription.

Function: `vaultRedemption(uint256 amount)`

Vault-specific redemption with immediate processing.

Inputs

- `amount`
  - **Control**: Full control.
  - **Constraints**: The caller must have `VAULT_ROLE`, `subscriptionPaused` must be false, and it can only be called by whitelisted vaults.
  - **Impact**: Amount of iUSPC tokens to redeem.

Branches and code coverage

Intended branches

- ☑ Whitelisted vaults can redeem and receive USDC immediately.
- ☑ After setting the redemption fee, verify that `_getRedemptionFees` and `$.collateral.safeTransferFrom` deduct the fee correctly, and the vault will receive the deducted USDC.

Negative behavior

- ☑ Revert if the caller is not `VAULT_ROLE`.
- ☑ Revert if the caller is not whitelisted.
- ☑ Revert if vault is paused.
- ☐ Verify whether the price returned by `pricer.getLatestPrice` has expired.

Function: `vaultSubscription(uint256 amount)`

Vault-specific subscription with immediate processing.

Inputs

- `amount`
  - **Control**: Full control.
  - **Constraints**: The caller must have `VAULT_ROLE`, `subscriptionPaused` must be false, and it can only be called by whitelisted vaults.
  - **Impact**: Amount of collateral to deposit

Branches and code coverage

Intended branches

- ☑ Whitelisted vaults can subscribe and receive iUSPC immediately.
- ☑ After setting the mint fee, the iUSPC actually received by the vault has already deducted the fee.
- ☑ This function can only be successfully called and the expected iUSPC will be received after the vault is set to `VAULT_ROLE` and is a whitelisted vault.

Negative behavior

- ☑ Revert if the caller is not `VAULT_ROLE`.
- ☑ Revert if the caller is not whitelisted.
- ☑ Revert if vault is paused.
- ☐ Verify whether the price returned by `pricer.getLatestPrice` has expired.

4.2. Module: USPC.sol

Function: `_deposit(address _caller, address _receiver, uint256 _assets, uint256 _shares)`

This function overrides ERC4626 `_deposit` to add donation attack protection.

Inputs

- `_caller`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: People whose underlying assets were taken away.

- `_receiver`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: People who receive shares.
- `_assets`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: The amount of underlying token managed by the Vault. Has units defined by the corresponding EIP-20 contract.
- `_shares`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: The amount of tokens in the vault. Has a ratio of underlying assets exchanged on mint/deposit/withdraw/redeem (as defined by the Vault).

Branches and code coverage

Intended branches

- ☑ Ensure that assets are safely transferred from the caller to the vault and mint an equivalent share for the recipient.
- ☑ When multiple different users make consecutive deposits, the vault status remains unchanged and the shares are not misallocated.
- ☑ The exchange rate calculation was accurate across various deposit amounts, and the overall exchange ratio remained stable.

Negative behavior

- ☑ Revert if the vault is paused.
- ☐ Revert if a small non-zero amount of shares remain after `_deposit`.

Function: `_update(address from, address to, uint256 value)`

This function is used to block transfers from/to denylisted addresses.

Inputs

- `from`
    - **Control**: Full control.
    - **Constraints**: It cannot be address 0, and it cannot be denylisted addresses.
    - **Impact**: The address sending tokens.

- `to`
    - **Control**: Full control.
    - **Constraints**: It cannot be address 0, and it cannot be denylisted addresses.
    - **Impact**: The address receiving tokens.
- `value`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: The amount of tokens being transferred.

Branches and code coverage

Intended branches

- ☑ Addresses not listed in the denylisted addresses can correctly complete the changes to balances and total amounts.

Negative behavior

- ☑ Revert if sender or receiver is on the transfer deny list
- ☐ Revert if contract is paused.

Function: `_withdraw(address _caller, address _receiver, address _owner, uint256 _assets, uint256 _shares)`

This function overrides ERC4626 `_withdraw` to add donation attack protection.

Inputs

- `_caller`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: People whose shares were taken away.
- `_receiver`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: People who receive assets.
- `_owner`
    - **Control**: Full control.
    - **Constraints**: Must equal `_caller` or grant sufficient allowance for the burn.
    - **Impact**: The account whose shares are burned and whose allowance may be

consumed.

- `_assets`
  - **Control**: Full control.
  - **Constraints**: Cannot exceed vault liquidity and must match the share-to-asset exchange rate.
  - **Impact**: The amount of the underlying token used for the Vault for accounting, depositing, and withdrawing.
- `_shares`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: The amount of tokens in the vault. Has a ratio of underlying assets exchanged on mint/deposit/withdraw/redeem (as defined by the Vault).

Branches and code coverage

Intended branches

- ☑ Burn the requested `_shares` from `_owner`, consuming allowance when `_caller != _owner`, and transfer the matching `_assets` to `_receiver`.
- ☑ Keep the share-to-asset exchange rate stable across sequential withdrawals so that no user can exit with more than their proportional claim.
- ☑ Enforce the donation-attack guard by ensuring `_shares` maps 1:1 to `_assets` once the withdrawal completes.

Negative behavior

- ☑ Revert if the vault is paused or withdrawals are globally disabled.
- ☑ Revert if rounding would leave residual shares or assets that break the donation-attack invariant.
- ☑ Revert if a small non-zero amount of shares remain after `_withdraw`.

5. Assessment Results

During our assessment on the scoped Multisafe USPC Contracts contracts, we discovered six findings. One critical issue was found. Three were of medium impact and two were of low impact.

5.1. 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.