our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Colb Finance

USC Engine

SECURITY REVIEW

Date: 2 October 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Colb Finance - USC Engine

Colb is the first native non-custodial tokenization solution that enables peerless access to Swiss-grade wealth management strategies, pre-IPO opportunities, and premium investment funds. It offers a bankruptcy-remote Trust structure and native ownership of real-world assets, all on-chain. Colb reduces the entrance threshold to such investments by removing constraints such as the minimum investment amount to get exposure to them. The protocol is designed with security at its core, boasting compliance with Swiss regulations and DeFi composability. Colb envisions a future rooted in transparency where every individual has equitable access to premium RWA investments.

Overview

The Engine contract serves as the core atomic minting vehicle for the Colb ecosystem, enabling users to mint $USC stablecoins by depositing whitelisted stable assets. The Engine operates as the primary interface between users holding other stablecoins and the $USC ecosystem, facilitating the conversion of various stable assets into $USC tokens through an atomic minting process that ensures price stability.

The system supports dynamic asset management, allowing administrators to onboard new stable assets, configure price oracles, and adjust operational parameters without requiring contract upgrades. Through its integration with external Oracle adapters and Whitelist contract, the Engine maintains real-time pricing while ensuring that only verified users can participate in minting operations under specific limits to ensure optimal system management.

4. Risk Classification

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable - and covers vectors similar to griefing attacks that can be easily repaired

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 3 days, with a total of 48 hours dedicated to the audit by two researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying four Medium and two Low severity issues. They are mainly related to incorrect collateral calculation, too high price deviation and others.

The Colb Finance team has done a great job with their test suite and provided support and responses to all of the questions that the Shieldify researchers had.

5.1 Protocol Summary

Project Name	**Colb Finance - USC Engine**
Repository	SmartContracts
Type of Project	RWA, Pre-IPO, USC Engine
Audit Timeline	3 days
Review Commit Hash	dbcc8a6e70ad0b3de34404117f03c47ff917a756
Fixes Review Commit Hash	5e1207f3e49473aab2182ad28b307771fa2c95c6

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
contracts/ramp/Engine.sol	237
contracts/compliance/RoleManager.sol	45
interfaces/ramp/IEngine.sol	14
interfaces/compliance/IRoleManager.sol	6
contracts/utils/RoleChecker.sol	11
Total	**313**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Medium** issues: **4**
- **Low** issues: **2**

ID	Title	Severity	Status
[M-01]	Incorrect Collateral Calculation Lets Users Mint Under-Collateralized USC	Medium	Fixed
[M-02]	Max Allowed Deviation of 500 (5%) Is Too High for `Engine` Logic	Medium	Acknowledged
[M-03]	Permit Signatures Fail Due to Oracle Price Changes Between Signature and Execution	Medium	Fixed
[M-04]	Account-Specific Zero Fee Setting Impossible Due to Fallback Logic	Medium	Acknowledged
[L-01]	Fee Calculation Uses Integer Division Rounding Down, Causing Protocol Revenue Loss	Low	Fixed
[L-02]	The `removeAsset()` Function in `Engine` Contract Emits `AssetSet` Instead of a Dedicated `AssetRemoved` Event	Low	Fixed

7. Findings

[M-01] Incorrect Collateral Calculation Lets Users Mint Under-Collateralized USC

Severity

Medium Risk

Description

The core pricing math in the `_getNormalizedAssetAmount()` function in the `Engine` contract is inverted:

`previewMintAccount()` -> `_getNormalizedAssetAmount()` uses multiplication by the oracle price instead of division:

```
// unitPrice = USD per 1 asset (1e18)
// BUG:
uint256 value = (uscAmount * unitPrice) / 1e18; // <-- should divide by
    price
```

Given price `p = USD/asset`, correct math is `assetNeeded = uscAmount * 1e18 / p`. Current code computes `assetCollected = uscAmount * p / 1e18`.

Impact:

- If `p < 1` (USDC/USDT below peg but within allowed deviation), the engine **under-collects** collateral.
- Under-collection fraction = **1 - p²** (e.g., p=0.995 ~0.9975% short; p=0.95 ~9.75% short if you ever allow 5%).
- If `p > 1`, honest users are **over-charged**.
- With the cap auto-increase on, the under-collateralized supply growth compounds.

This is independent of fees; fees are applied to the already miscomputed `assetAmount`.

Scenarios

First, the audit specs documentation says that the accepted assets for the `Engine` smart contract will only be USDC and USDT.

What the code does:

Inputs (example)

- You call `flashMintWithPermit()` (or `flashMint()`) with:
  - `amount = 100_000 * 10^6` (mint **100,000 USC**; USC has 6 dp)
  - `asset = USDC` (6 dp)
- The engine computes:
  1. `unitPrice = getAssetToUSDUnitPrice(USDC)`

     oracle returns **USD per 1 USDC** scaled to 1e18
     - `$1.00` `1e18`
     - `$0.995` `0.995e18`
     - `$0.99` `0.99e18`
     - `$1.05` `1.05e18`
  2. `assetAmount = _getNormalizedAssetAmount(uscAmount, unitPrice, 6)`

Inside `_getNormalizedAssetAmount()`:

```
uint256 value = (uscAmount * unitPrice) / 1e18;    // -< WRONG OPERATION
// adjust to asset decimals (6 == USC_DECIMALS, so no change)
return value;
```

So the engine collects:

```
assetCollected_code = uscAmount * price
```

(where `uscAmount` is treated like USD, and `price` is USD/asset).

That's multiplication.

What it should do (correct math)

We want:

```
assetNeeded_true = USD_to_raise / (USD_per_asset)
                 = uscAmount / price
```

That's division.

Concrete numbers (USDC, 6 dp)

Lets mint 100,000 USC. Below, we can see what the code collects vs what it should collect.

Oracle price (USD/USDC)	Allowed by max default 500 bps?	Collected by code (= A × p)	Correct (= A ÷ p)	Shortfall vs correct
1.0000	yes	100,000.00	100,000.00	0.00 (0.00%)
0.9950	yes (lower bound)	99,500.00	100,502.5126	1,002.5126 (0.9975%)
0.9900	yes ($\geq$1% tolerance)	99,000.00	101,010.1010	2,010.1010 (1.99%)
0.9500	yes (5% tolerance)	95,000.00	105,263.1579	10,263.1579 (9.75%)
1.0500	yes (5% tolerance)	105,000.00	95,238.0952	user overpays 9,762 (10.25%)

For example: - with `50 bps` (±0.5%), the code allows price `0.995-1.005` and at the worst allowed sub-peg (0.995), the code under-collects by **~1.0%** (not 0.5%). Reason: error factor is p^2. See below. - with `500 bps` (5%) and oracle price of `$0.95`, the code under-collects by \codex{~9.75%} per mint.

So even with a minimum \codex{0.5%} deviation band, the code is leaking \codex{~1.0%} per mint at the lower edge.

Location of Affected Code

File: contracts/ramp/Engine.sol#L420

```
/**
 * @inheritdoc IEngine
 */
function previewMintAccount(uint256 amount, address asset, address
   account) public view returns (uint256 assetAmount, uint256 fee) {
    uint256 accFeeRate = accountFeeRate[account];
    // set asset-specific fee rate to use if custom is not set
    if (accFeeRate == 0) accFeeRate = assetConfigs[asset].feeRate;
    // get normalized amount in asset decimals
    assetAmount = _getNormalizedAssetAmount(amount,
       getAssetToUSDUnitPrice(asset), assetConfigs[asset].decimals);
    // calculate fee based on asset amount and fee rate
    fee = accFeeRate > 0 ? (assetAmount * accFeeRate) / BPS_DENOMINATOR :
         0;
}
```

File: contracts/ramp/Engine.sol#L477

```
/**
 * @notice Calculates the normalized asset amount for minting a given
    $USC amount.
 */
function _getNormalizedAssetAmount(uint256 uscAmount, uint256 unitPrice,
   uint8 decimals) internal pure returns (uint256) {
    // @note: the `unitPrice` has precision of 18
    uint256 value = (uscAmount * unitPrice) / SCALE;
    // normalize to asset decimals
    if (decimals > USC_DECIMALS) {
        return value * 10**(decimals - USC_DECIMALS);
    } else if (USC_DECIMALS > decimals) {
        return value / 10**(USC_DECIMALS - decimals);
    }
    // return `value` if decimals match
    return value;
}
```

Recommendation

Fix the formula to divide by the USD/asset price and round up in the `_getNormalizedAssetAmount()` function in the `Engine` contract:

```
// 6dp USC → asset units (still 6dp), then rescale to asset decimals
uint256 assetUnits6 = Math.mulDiv(uscAmount, 1e18, unitUsdPrice1e18, Math
   .Rounding.Up);
// rescale to asset decimals (existing logic)
```

Team Response

Fixed.

[M-02] Max Allowed Deviation of 500 (5%) Is Too High for `Engine` Logic

Severity

Medium Risk

Description

Excessive price-deviation ceiling (±5%) enables loss and manipulation.

`MAX_PRICE_DEVIATION_UPPER_BOUND = 500` lets operators configure assets to mint when the oracle reads **$0.95–$1.05** for a $1 stable (USDC/USDT). That's unjustifiably wide for USDC/USDT, dangerous and basically material depeg. If the oracle is nudged to `$1.05` (allowed), minters can pay **~4.76% fewer tokens** (1/1.05). Basically, a ±5% gate is a big manipulation window. Allowing mints in this window creates avoidable P&L exposure and a clear gaming surface: - The `Engine` smart contract continues minting while the stable trades at `$0.95` or `$1.05`. If/when price mean-reverts, the protocol is left with an adverse inventory P&L versus newly minted USC.

For dollar stables (in the audit specs documentation, it states that the accepted assets will only be USDC and USDT), a 5% allowance is operationally reckless. If a collateral stable (USDC/USDT) is off-peg meaningfully, stop minting USC. Don't keep issuing ("printing") new USC while the reference asset is broken.

Location of Affected code:

File: contracts/ramp/Engine.sol#L52-L57

```
/**
 * @notice Maximum allowed price deviation upper bound from 1 USC (5%)
 */
uint256 public constant MAX_PRICE_DEVIATION_UPPER_BOUND = 500; // 5% in
   basis points

/**
 * @notice Maximum allowed price deviation from 1 USC (0.5%)
 */
uint256 public constant DEFAULT_MAX_PRICE_DEVIATION = 50; // 0.5% in
   basis points
```

File: contracts/ramp/Engine.sol#L511

```
/**
 * @notice Validates that the oracle price is within acceptable deviation
     limits.
 */
function _validatePriceDeviation(uint256 price, address asset) internal
   view {
    uint256 maxDeviation = assetConfigs[asset].maxDeviation;
    // calculate deviation tolerance value
    uint256 deviationTolerance = (SCALE * maxDeviation) / BPS_DENOMINATOR
       ;
    // get upper bound
    uint256 upperBound = SCALE + deviationTolerance;
    // get lower bound
    uint256 lowerBound = SCALE - deviationTolerance;
    // validate the price deviation
    if (price < lowerBound || price > upperBound) revert
       PriceDeviationExceeded(price, lowerBound, upperBound);
}
```

Recommendation

Set upper bound (`MAX_PRICE_DEVIATION_UPPER_BOUND`): `100 bps`. There is no good operational reason to ever allow 5% on a "stable". If there is a real 5% move, you should pause or de-whitelist the asset, not keep minting.

Consider the following changes: - Default per-asset deviation: `USDC/USDT: 25-50 bps` - DAI/FRAX/LUSD (softer/algorithmic): `50-150 bps` (only if you must)

Team Response

Acknowledged.

[M-03] Permit Signatures Fail Due to Oracle Price Changes Between Signature and Execution

Severity

Medium Risk

Description

The `flashMintWithPermit()` function contains a critical flaw where ERC-2612 permit signatures are likely to fail due to a timing mismatch between signature creation and transaction execution.

The permit signature includes a specific amount parameter that represents the total tokens to be transferred (`assetAmount` + `fee`). However, this amount is calculated using Oracle prices:

- At signature time: User calculates `fullPayout` using the current oracle price and signs the permit
- At execution time: Contract recalculates `fullPayout` using potentially different oracle price

- Result: Amounts don't match, permit call fails

```
// Amount calculated at execution time
(uint256 assetAmount, uint256 fee) = previewMint(amount, asset);
uint256 fullPayout = assetAmount + fee;

// Permit expects amount from signature time
try IERC20Permit(asset).permit(msg.sender, address(this), fullPayout,
    deadline, v, r, s) {} catch {}
```

The `previewMint()` function calls `getAssetToUSDUnitPrice()`, which fetches fresh oracle data, making the calculated amount unpredictable between signature and execution.

Location of Affected Code

File: contracts/ramp/Engine.sol#L338

```
// Line ~330: Amount calculated at execution time
(uint256 assetAmount, uint256 fee) = previewMint(amount, asset);
uint256 fullPayout = assetAmount + fee;

// Line ~335: Permit expects amount from signature time
try IERC20Permit(asset).permit(msg.sender, address(this), fullPayout,
    deadline, v, r, s) {} catch {}
```

Impact

The permit feature becomes essentially unusable and most permit-based transactions will fail silently.

Recommendation

Modify the function to accept the expected permit amount as a parameter.

Team Response

Fixed.

[M-04] Account-Specific Zero Fee Setting Impossible Due to Fallback Logic

Severity

Medium Risk

Description

The fee calculation logic in `previewMintAccount()` makes it impossible to set a zero fee for specific accounts when the asset has a non-zero default fee rate. The code treats `accountFeeRate[account] = 0` as "not set" and falls back to the asset's fee rate, preventing operators from exempting specific accounts from fees.

The code cannot distinguish between:

- Account fee not configured (should use asset default)
- Account fee explicitly set to zero (should charge no fees)

Both cases result in `accFeeRate == 0`, so the fallback logic always applies.

Location of Affected Code

File: contracts/ramp/Engine.sol#L420

```
function previewMintAccount(uint256 amount, address asset, address
    account) public view returns (uint256 assetAmount, uint256 fee) {
  uint256 accFeeRate = accountFeeRate[account];
  // set asset-specific fee rate to use if custom is not set
  if (accFeeRate == 0) accFeeRate = assetConfigs[asset].feeRate;
  // code
}
```

Impact

Protocol operators cannot waive fees for a specific set of users.

Recommendation

Modify the storage pattern to distinguish between "not set" and "explicitly zero".

Team Response

Acknowledged.

[L-01] Fee Calculation Uses Integer Division Rounding Down, Causing Protocol Revenue Loss

Severity

Low Risk

Description

The fee calculation in `previewMintAccount()` uses standard integer division which always rounds down in Solidity. This causes the protocol to systematically lose revenue when fees don't divide evenly.

In Solidity, integer division truncates (rounds down) any remainder. For fee calculations, this should always round UP to ensure the protocol collects the full intended fee amount.

Location of Affected Code

File: contracts/ramp/Engine.sol#L427

```
function previewMintAccount(uint256 amount, address asset, address
    account) public view returns (uint256 assetAmount, uint256 fee) {
  // code
  // @audit Always rounds DOWN due to integer division
  fee = accFeeRate > 0 ? (assetAmount * accFeeRate) / BPS_DENOMINATOR :
     0;
}
```

Impact

Every transaction with non-zero remainder loses fee income for protocol.

Recommendation

Replace the current fee calculation with proper rounding up.

Team Response

Fixed.

[L-02] The `removeAsset()` Function in `Engine` Contract Emits `AssetSet` Instead of a Dedicated `AssetRemoved` Event

Severity

Low Risk

Description

The `removeAsset()` function emits `AssetSet(asset, zeroedConfig)` instead of a dedicated `AssetRemoved` event. Additionally, the interface declares events (`AssetRemoved` , `AssetOracleUpdated` , `AssetPriceDeviationUpdated` , `AssetPermitSupportUpdated` , `FeeRateUpdated` , `USCUpdated`) that the implementation never emits. Off-chain indexers/alerts wired to the interface events will miss removals and field-level changes.

Location of Affected code:

File: contracts/ramp/Engine.sol#L214

```
/**
 * @inheritdoc IEngine
 */
function removeAsset(address asset) external {
    RoleChecker.hasRole(roleManager, ENGINE_ASSET_MANAGER_ROLE);

    if (asset == address(0)) {
        revert ZeroAddress();
    }

    delete assetConfigs[asset];

    emit AssetSet(asset, assetConfigs[asset]);
}
```

Recommendation

Emit `AssetRemoved(asset)` in `removeAsset()`.

Team Response

Fixed.

our shielding · Your smart contracts, our shielding · Your smart

shieldify

Thank you!