yAudit SOFA Review

Review Resources:

- Automator 2.0 & Dual Currency Vault documentation

Auditors:

- Drastic Watermelon
- adriro

Table of Contents

1 Review Summary

2 Scope

3 Code Evaluation Matrix

4 Findings Explanation

5 Critical Findings

  a 1. Critical - Token ID collision enables vault drainage

6 High Findings

  a 1. High - `CrvUSDDualVault.harvest()` calculates the amount of `crvUSD` to be withdrawn incorrectly

7 Medium Findings

  a 1. Medium - Signature replay protection is ineffective against contract signatures

  b 2. Medium - Rounding errors can lead to minor insolvency issues

8 Low Findings

  a 1. Low - Missing reentrancy guard in CrvUSDDualVault.sol

  b 2. Low - EIP712 domain separator isn't protected from chain forks

c 3. Low - Missing vault whitelist check in `CrvUSDAutomatorBase.burnProducts()`
9 Gas Saving Findings
a 1. Gas - Optimize for loops
10 Informational Findings
a 1. Informational - Incorrect arguments in `FeeCollected` event
b 2. Informational - Unnecessary native payments implementation
c 3. Informational - Use ERC20Upgradeable in CrvUSDAutomatorBase.sol
d 4. Informational - Usage of outdated OpenZeppelin libraries
11 Final Remarks

Review Summary

Automator

The crvUSD Automator is a type of vault in which users deposit crvUSD, which is re-deposited into scrvUSD for passive yield. Strategists, called Optivisors, use these funds to purchase structured products from SOFA Vaults to maximize capital.

Dual Currency Vault

Dual Currency Vault is a structured product designed by inheriting the SOFA Vault's design principles, using the ERC1155 standard, and implementing it on-chain. Users set an expiration date, anchor price, and the amount of funds they wish to deposit. Market makers will provide quotes for the product, and the user's funds and the premium provided by the market maker will be transferred to the vault. The market maker can exchange the asset for the specified anchor price from the vault before 2 hours after the expiration date (American Option). After the expiration + 2 hours, users can burn their position to receive their funds and earnings.

The contracts of the SOFA repository were reviewed over 5 days. Two auditors performed the code review between 20th January and 24th January 2024. The repository was under active development during the review, but the review was limited to the latest commit 2b45e2bd01141cb6dd8302f8ea8579de0fbc6c9e of the SOFA repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
contracts/
├── automators
│   ├── bases
│   │   └── CrvUSDAutomatorBase.sol
│   └── vaults
│       └── SimpleSmartTrendVault.sol
├── mocks
│   └── automators
│       └── CrvUSDAutomatorFactory.sol
└── vaults
    └── CrvUSDDualVault.sol
```

After the findings were presented to the SOFA team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, SOFA and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Correct access control usage is made within the contracts.
Mathematics	Low	Although simple math relations are used within the in-scope contracts, operations with incorrect rounding directions were identified.

Category	Mark	Description
Complexity	Average	The contracts integrate with Curve's `scrvUSD` vault to generate yield for the protocol and its users.
Libraries	Average	The protocol relies heavily on OpenZeppelin's libraries, which were found to be utilized correctly. Additionally, the libraries may be utilized for tasks executed by the protocol's contracts.
Decentralization	Good	The in-scope contracts require little intervention from privileged actors that aren't necessarily tied to the protocol's team.
Code stability	Good	The code repository wasn't actively developed during the review.
Documentation	Low	The methods within the in-scope contracts seldom present in-line comments or NATSPEC documentation, which would improve the contracts' readability.
Monitoring	Average	The log coverage is acceptable except for one incorrect parameter.
Testing and verification	Low	The provided contracts have no unit tests. Developing a thorough testing suite for the in-scope contracts is strongly advised.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational

- Findings including recommendations and best practices.

Critical Findings

1. Critical - Token ID collision enables vault drainage

A collision in how token IDs are generated can be maliciously used by the maker to remove the minter's assets.

Technical Details

The implementation of CrvUSDDualVault.sol uses two different ways of generating the corresponding tokens IDs to minter and maker.

```
    function getProductId(uint256 expiry, uint256 anchorPrice, uint256 isMaker) public pure returns (uint256) {
        return uint256(keccak256(abi.encodePacked(expiry, anchorPrice, isMaker)));
    }

    function getMinterProductId(uint256 expiry, uint256 anchorPrice, uint256 premiumPercentage) public pure returns (uint256) {
        return uint256(keccak256(abi.encodePacked(expiry, anchorPrice, premiumPercentage)));
    }

```

The minter gets a token with `ID=keccak(expiry, anchorPrice, premiumPercentage)`, and the maker gets one with `ID=keccak(expiry, anchorPrice, 1)`. By setting a `premiumPercentage` value of 1, their hashes collide, leading to both IDs being the same.

The `burn()` function takes the premium percentage as a user-supplied argument. A malicious maker can call this function with `premiumPercentage = 1` pretending to be the minter, an action that would be allowed since the token ID will result in the same value as the maker token ID.

This way, an attacker can steal the minter's collateral and quote assets by calling `burn()` after maturity. The attacker can also create a fake agreement between a minter and a maker they control to withdraw the provided assets twice, draining the vault.

Proof of Concept

```
function test_dualVault_collision() public {
    uint256 totalCollateral = 1_000e18;
    uint256 expiry = block.timestamp + 14 days;
    uint256 makerCollateral = 500e18;
    uint256 anchorPrice = 3_000e8;
    uint256 premiumPercentage = makerCollateral * 1e18 / totalCollateral;

    deal(crvUSD, user, totalCollateral - makerCollateral);
    deal(crvUSD, maker, makerCollateral);

    CrvUSDDualVault.MintParams memory mintParams = CrvUSDDualVault.MintParams({
        expiry: expiry,
        anchorPrice: anchorPrice,
        makerCollateral: makerCollateral,
        deadline: expiry,
        maker: maker,
        makerSignature: ""
    });

    vm.prank(user);
    dualVault.mint(totalCollateral, mintParams, address(0xdead));

    vm.warp(expiry + 2 hours);

    // Maker steals collateral by calling burn
    (bool success,) = Maker(maker).execute(
        address(dualVault),
        abi.encodeWithSelector(
            CrvUSDDualVault.burn.selector,
            expiry,
            mintParams.anchorPrice,
            1
        )
    );
    assertTrue(success, "failed to call burn");

    // maker has all the collateral
    assertEq(IERC20(crvUSD).balanceOf(maker), totalCollateral);

    // burn will fail here, vault is insolvent
    vm.expectRevert("insufficient shares to redeem");
    vm.prank(user);
    dualVault.burn(expiry, anchorPrice, premiumPercentage);
}
```

Impact

Critical. The attacker can drain assets from existing agreements.

Recommendation

The `CrvUSDDualVault` contract ought to differentiate positions assigned to minters or market makers by using an `isMaker` flag within the product id calculation, similarly to how `SimpleSmartTrendVault.getProductId` does.

To achieve this, the contract could modify its `getProductId` method as follows:

```diff
-    function getProductId(uint256 expiry, uint256 anchorPrice, uint256 isMaker) public pure returns (uint256) {
-        return uint256(keccak256(abi.encodePacked(expiry, anchorPrice, isMaker)));
+    function getProductId(uint256 expiry, uint256 anchorPrice, uint256 premiumPercentage, uint256 isMaker) public pure returns (uint256) {
+        return uint256(keccak256(abi.encodePacked(expiry, anchorPrice, premiumPercentage, isMaker)));
```

This way, usages of the method can distinguish between minter and maker positions, preventing the shown collision from occurring.

Developer Response

Fixed in the commit 2d72ad1abd940d46675921cad07454f85c7e03ab.

High Findings

1. High - `CrvUSDDualVault.harvest()` calculates the amount of `crvUSD` to be withdrawn incorrectly

`CrvUSDDualVault.harvest()` calculates `fee = scrvUSD.balanceOf(address(this)) - totalDeposit`, which is incorrect as it's subtracting amounts denominated in different tokens.

Technical Details

`scrvUSDBalanceOf(address(this))` is an amount of `scrvUSD` tokens, while `totalDeposit` is an amount of `crvUSD` tokens as amounts of `collateral` are added to it within the `_mint()` and `_mintBatch()`.

Given that YearnV3 vault shares, like `scrvUSD`, accrue value regarding the vault's asset, the contract's `scrvUSD` balance will not increase to reflect the yield accrued by such a position. As a result, the contract fails to account and withdraw its generated fees.

Impact

High. The `CrvUSDDualVault` fee withdrawal flow will malfunction.

Recommendation

The highlighted method should consider the fee accrued as the difference between the value of its `scrvUSD` shares in `crvUSD` and `totalDeposit`, to denominate the calculation entirely in `crvUSD`:

```
interface IScrvUSD {
     function deposit(uint256 assets, address receiver) external returns (uint256
shares);
     function withdraw(uint256 assets, address receiver, address owner) external returns
(uint256 shares);
     function approve(address spender, uint256 amount) external returns (bool);
+    function previewRedeem(uint256 shares) external view returns(uint256 assets);
 }


     function harvest() external {
-        uint256 fee = scrvUSD.balanceOf(address(this)) - totalDeposit;
+        uint256 balance = scrvUSD.balanceOf(address(this));
+        uint256 fee = scrvUSD.previewRedeem(balance) - totalDeposit;
         require(fee > 0, "Vault: zero fee");
         scrvUSD.withdraw(fee, feeCollector, address(this));

     function totalFee() external view returns (uint256) {
-        return scrvUSD.balanceOf(address(this)) - totalDeposit;
+        uint256 balance = scrvUSD.balanceOf(address(this));
+        return scrvUSD.previewRedeem(balance) - totalDeposit;
     }
```

Developer Response

Fixed in the commit c013a2d734565d61def89aa8d728420a082246de.

Medium Findings

1. Medium - Signature replay protection is ineffective against contract signatures

The current signature replay protection could lead to different issues when combined with EIP-1271 signatures.

Technical Details

Signature replay protection is given the SignatureBitMap.sol mixin. Hashed signatures are stored in a mapping and later recovered to check if the signature has been previously used.

While EOA signature malleability is covered by OpenZeppelin's ECDSA library, the implementation also supports EIP-1271 signatures as it uses the SignatureChecker library, which can deal with both EOA and contract signatures.

In the case of contract signatures, malleability protection would depend on the implementation of EIP-1271 for each specific smart account. For example, the Safe wallet implementation checks the signature payload has at least the required length to match the number of signatures but doesn't impose any other restriction over the length (see `checkNSignatures()` implementation). A malicious user can reuse the signature by padding the payload with dummy data to generate a different hash.

Contract signatures enable an additional vector attack by maliciously consuming a signature before it is rightfully used. An attacker can deploy a contract whose `isValidSignature()` returns ok for any payload and then submit a transaction to the vault using this contract as the maker and the signature they want to invalidate.

Impact
Medium. Although not frequently used, contract signatures could be replayed using the current design. Additionally, an attacker can intentionally grieve other valid EOA signatures by front-running the transaction and consuming the signature before it is used.

Recommendation
Use nonces for replay protection or disable EIP-1271 signatures.

Developer Response
As long as the maker agrees, it is not unacceptable for users to reuse their signatures. Neither the protocol nor the users suffer any loss. The scenario of "reuse the signature by padding the payload with dummy data to generate a different hash" falls strictly under this case.

For "maliciously consuming a signature before it is rightfully used," the attacker gains no real benefit. Moreover, the user can request a new signature to proceed with the transaction.

We support EIP-1271 to accommodate contract users (e.g., multisig wallets) who wish to act as makers. We avoid using a nonce mechanism similar to blockchain transactions because the maker cannot effectively determine which quotes have been submitted on-chain and which have not, making it difficult to assign a meaningful nonce value.

2. Medium - Rounding errors can lead to minor insolvency issues
The collateral and quote payoff calculations in CrvUSDDualVault.sol use an incorrect rounding, leading to small amount differences that could ultimately prevent redeeming the full minted amounts.

Technical Details
When the position is redeemed, the implementation of `_burn()` calculates the corresponding amount of collateral and quote assets that should be transferred to the user.

```
        collateralPayoff = amount - amount * makerPosition / totalPosition;
        quoteAssetPayoff = (amount - collateralPayoff) * anchorPrice *
quoteAsset.decimals() / collateral.decimals() / PRICE_DECIMALS;
```

`collateralPayoff` is the amount minus the user's share of exchanged collateral, while `quoteAssetPayoff` is the user's share of exchanged collateral projected in the quote asset domain.

Note that `collateralPayoff` is rounded up. Since `amount * makerPosition / totalPosition` rounds down, the subtraction `amount - amount * makerPosition / totalPosition` will be rounded up.

Additionally, note that if `collateralPayoff` is fixed, this will also impact the calculation of `quoteAssetPayoff`, which would incorrectly round up due to the subtraction of `amount - collateralPayoff`.

The same issue is present in the batch variant of the burn functionality.

Impact

Medium. Incorrect rounding direction could generate small token insolvency issues.

Recommendation

The calculation of `collateralPayoff` should round up the term `amount * makerPosition / totalPosition`.

```
collateralPayoff = amount - Math.mulDiv(amount, makerPosition, totalPosition,
Math.Rounding.UP);
```

The calculation of `quoteAssetPayoff` can be simplified as:

```
quoteAssetPayoff = (amount * makerPosition / totalPosition) * anchorPrice *
quoteAsset.decimals() / collateral.decimals() / PRICE_DECIMALS;
```

Developer Response

Fixed in the commit b31276758289715a1c367bd459cf6be9b33117d0.

Low Findings

1. Low - Missing reentrancy guard in CrvUSDDualVault.sol

Some functions in the contract don’t implement reentrancy protection.

Technical Details

- `mint()`
- `mintBatch()`

- `harvest()`

Impact

Low.

Recommendation

Add the `nonReentrant` modifier.

Developer Response

Fixed in the commit c917dd5675cf95dcc84e1737bdbd9155f44dcff3.

2. Low - EIP712 domain separator isn’t protected from chain forks

`SimpleSmartTrendVault` and `CrvUSDDualVault` both store their EIP712 domain separators as constants: if the deployment chain were to fork into two separate chains, signatures from the original chain would be valid on the child chain as well.

Technical Details

Both contracts calculate their `DOMAIN_SEPARATOR` field during a call to `initialize()`. Because such field cannot be modified, the calculated separator will be stored in the proxy’s storage as a constant: 1 and 2

In the event of a fork to the chain on which the contracts are deployed, if no action is taken to modify such value via an implementation upgrade, both versions of the contracts will consider signatures signed for the original chain as valid.

Impact

Low.

Recommendation

Use OpenZeppelin’s EIP712 library to execute all EIP712-related logic.

Developer Response

Acknowledged.

3. Low - Missing vault whitelist check in `CrvUSDAutomatorBase.burnProducts()`

`CrvUSDAutomatorBase.burnProducts()` fails to verify that each vault it interacts with has been whitelisted within the associated `AutomatorFactory`.

Technical Details

While `CrvUSDAutomatorBase.mintProducts()` correctly verifies that all vaults are whitelisted, the `CrvUSDAutomatorBase.burnProducts()` method fails to do so.

Impact

Low.

Recommendation

Implement the whitelist check used in `mintProducts()` also within `burnProducts()`.

Developer Response

`burnProducts()` does not need to check vaults; positions in non-whitelisted vaults can also be burned, and the proceeds will be counted as revenue.

Gas Saving Findings

1. Gas - Optimize for loops

For loops across all contracts in scope can be optimized to reduce their gas consumption.

Technical Details

For loop, gas consumption can be reduced by:

1. Avoiding to initialize the iteration variable to its default value: `for (uint i = 0; ..; ..)` should be replaced with `for (uint i; ..; ..)`
2. Caching an array's length in a variable:

```
for(uint i; i < array.length; i++)
```

should be replaced with

```
uint256 len = array.length;
for(uint i; i < len; i++)
```

Impact

Gas optimization.

Recommendation

Implement the suggested modifications to reduce the contracts' gas consumption.

Developer Response

Refactor for loop to save gas in the commit 10dab5df316a696f2fdb4b83c5ce8821f6838658.

Some parts were not optimized due to stack too deep issues after optimization. The existing deployed contracts will not be optimized for now to maintain code consistency.

Informational Findings

1. Informational - Incorrect arguments in `FeeCollected` event

The `fee` and `protocolFee` arguments are always set to zero.

Technical Details

The `harvest()` function emits the `FeeCollected` event after clearing the `totalFee` and `totalProtocolFee` variables.

```solidity
    function harvest() external nonReentrant {
        require(totalFee > 0 || totalProtocolFee > 0, "Automator: zero fee");
        uint256 feeAmount = 0;
        uint256 protocolFeeAmount = 0;
        if (totalFee > 0) {
            feeAmount = scrvUSD.redeem(uint256(totalFee), owner(), address(this));
            totalFee = 0;
        }
        if (totalProtocolFee > 0) {
            protocolFeeAmount = scrvUSD.redeem(totalProtocolFee, IAutomatorFactory(factory).feeCollector(), address(this));
            totalProtocolFee = 0;
        }
        emit FeeCollected(_msgSender(), feeAmount, totalFee, protocolFeeAmount, totalProtocolFee);
    }
```

Impact

Informational.

Recommendation

Store these values in a local variable. This will also improve gas usage.

```
function harvest() external nonReentrant {
    int256 totalFeeCached = totalFee;
    uint256 totalProtocolFeeCached = totalProtocolFee;

    require(totalFeeCached > 0 || totalProtocolFeeCached > 0, "Automator: zero fee");

    uint256 feeAmount = 0;
    uint256 protocolFeeAmount = 0;

    if (totalFeeCached > 0) {
        feeAmount = scrvUSD.redeem(uint256(totalFeeCached), owner(), address(this));
        totalFee = 0;
    }

    if (totalProtocolFeeCached > 0) {
        protocolFeeAmount = scrvUSD.redeem(totalProtocolFeeCached,
IAutomatorFactory(factory).feeCollector(), address(this));
        totalProtocolFee = 0;
    }

    emit FeeCollected(_msgSender(), feeAmount, totalFeeCached, protocolFeeAmount,
totalProtocolFeeCached);
}
```

Developer Response

Fixed in the commit b95f8188353e7dae0d656c49096e78284926aa84.

2. Informational - Unnecessary native payments implementation

The SimpleSmartTrendVault.sol and CrvUSDDualVault.sol contracts implement the `receive()` function despite not requiring native payments.

Technical Details

- SimpleSmartTrendVault.sol#L56
- CrvUSDDualVault.sol#L73

Impact

Informational.

Recommendation

Remove the `receive()` function.

Developer Response

Fixed in the commit 3bb5fdf85cb8a50a9e758cfd86a58c097fa94afe.

Only the Dual Vault has been updated. The SimpleSmartTrendVault is already running on-chain. To maintain code consistency and facilitate the reuse of deployed implementations for similar products, it will not be modified for now.

3. Informational - Use ERC20Upgradeable in CrvUSDAutomatorBase.sol

The contract follows a cloneable pattern, making the upgradeable variant of the library a better fit.

Technical Details

Using the ERC20Upgradeable library instead of ERC20 would allow to properly initialize the name and symbol properties of the token without needing to redefine these.

Impact

Informational.

Recommendation

Use ERC20Upgradeable as the base contract in CrvUSDAutomatorBase.sol.

Developer Response

Acknowledged.

4. Informational - Usage of outdated OpenZeppelin libraries

The protocol makes use of outdated version of the OpenZeppelin libraries.

Technical Details

The protocol uses the following versions of OpenZeppelin's libraries:

```
"@openzeppelin/contracts": "^4.7.3",
"@openzeppelin/contracts-upgradeable": "^4.8.0",
```

Which belong to outdated major and minor releases.

Impact

Informational.

Recommendation

The protocol should upgrade the libraries to the latest minor release within the `v4` major release: `v4.9.6`.

Developer Response

Acknowledged.

Final Remarks

Sofa's `crvUSD` automators and vaults allow users to purchase, sell and execute different financial options utilizing the `crvUSD` stablecoin. The contracts then rely on `scrvUSD` to generate a yield while the options reach maturity, granting rewards to the protocol and its users.

While the audited contracts are relatively similar to the protocol's automators and vaults, which utilize Aave's aTokens, a core difference in how such tokens and `scrvUSD` function generated an incorrect accounting issue in the protocol's fee redemption flow.

Additionally, within `CrvUSDDualVault`, calculating a product's ID without clear distinction between minter and maker tokens allows a malicious user to drain the vault of its `crvUSD` holdings entirely.

The team promptly fixed critical issues, while others were acknowledged but left unchanged to minimize code and contract structure modifications. Given the severity of the findings and the lack of tests, auditors strongly recommend building a testing suite that covers these contracts, particularly the new Dual Vault system.