code4rena

SOFA.org Automator

Smart Contract Security Assessment

Version 2.0

Audit dates: Oct 17 — Oct 25, 2024

Audited by: peakbolt
spicymeatball

Contents

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About SOFA.org Automator

SOFA.org is a decentralized, non-profit organization focused on advancing the DeFi ecosystem. Our modus operandi is to promote the highest DeFi standards, support high-quality projects, and promote adoption of blockchain technologies across mainstream finance.

2.2 Scope

Repository	sofa-org/sofa-protocol/blob/feat/auto/contracts/vaults/Automator.sol
Commit Hash	5f63be56ab094714fd44eab338007839a320b327
Mitigation Hash	0c389bcdfcb32c66b3e72c13f89880c292f6b848

2.3 Audit Timeline

DATE	EVENT
Oct 17, 2024	Audit start
Oct 25, 2024	Audit end
Oct 30, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	4
Medium Risk	2
Low Risk	1
Informational	0
Total Issues	7

3. Findings Summary

ID	DESCRIPTION	STATUS
H-1	Lack of \`msg.sender\` validation for \`onERC1155Received()\` and \`onERC1155BatchReceived()\`	Resolved

H-2	An attacker can pass arbitrary mint parameters by frontrunning `mintProducts()`	Resolved
H-3	`vault` parameter is not verified in `burnProducts()`	Resolved
H-4	Products for each `_positions[id]` can be partially burnt in `burnProducts()` to distort PnL calculation	Resolved
M-1	Users can frontrun losses after 7 days of depositing by requesting redemption upon deposit	Resolved
M-2	Automator will be bricked when `totalCollateral` reaches zero	Acknowledged
L-1	Harvesting fees can prevent depositors from claiming their collateral	Resolved

4. Findings

4.1 High Risk

A total of 4 high risk findings were identified.

[H-1] Lack of `msg.sender` validation for `onERC1155Received()` and `onERC1155BatchReceived()`

Severity: High	Status: Resolved

Context:

- Automator.sol#L43

Description:

Automator will receive product vault ERC1155 shares during `mintProducts()`, which will trigger `onERC1155Received()` and `onERC1155BatchReceived()`.

However, there is no validation of `msg.sender` during the ERC1155 callback to ensure that the product shares comes from the valid vaults.

Due to the issue, it is possible for a user to transfer product shares into Automator, and then burn it to perform a stealth donation as part of a share inflation attack.

The share inflation attack can occur as follows,

1. Attacker frontruns Victim with a 1 wei deposit (WETH) to receive 1 wei share.
2. Attacker then donates 1 WETH worth of product vault shares into Automator and call `burnProducts()`, which will then indirectly donate 1 WETH to Automator and inflate the share price for the 1 wei share held by Attacker.
3. Now totalCollateral = 1.000000000000000001, Victim will deposit 1 WETH but receives 0 shares.
4. Attacker will then profit and withdraw the 2 WETH + 1 wei.

Recommendation:

There are two possible solutions,

1. Perform whitelisting at `onERC1155Received()` and `onERC1155BatchReceived()` to ensure that the product shares are from valid vaults, not a malicious user. Note: transfers can only happen via `safeTransferFrom()` and `safeBatchTransferFrom()`, which will trigger the receive callback.

2. Increase the attack cost of share inflation attack by minting dead shares for the first deposit.

SOFA.org: The issue has been resolved with @0c389bc.

Zenith: Verified.

[H-2] An attacker can pass arbitrary mint parameters by frontrunning `mintProducts()`

Severity: High | Status: Resolved

Context:

- Automator.sol#L159-L162
- SignatureCheckerUpgradeable.sol#L35-L37

Description: When the `mintProducts` function is called, the automator only checks if the signatures in `mintParams.makerSignature` are from a whitelisted `maker`:

```
    function mintProducts(
        ProductMint[] calldata products,
        bytes calldata signature
    ) external {
        bytes32 signatures;
        for (uint256 i = 0; i < products.length; i++) {
            require(vaults[products[i].vault], "Automator: invalid
vault");
            ---SNIP---
            signatures = signatures ^
keccak256(products[i].mintParams.makerSignature);
        }

        (address signer, ) =
signatures.toEthSignedMessageHash().tryRecover(signature);
>>      require(makers[signer], "Automator: invalid maker");
```

Other parameters are left unchecked and are assumed to be verified inside the vault. For instance:

```
        // verify maker's signature
        bytes32 digest =
            keccak256(abi.encodePacked(
                "\x19\x01",
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(MINT_TYPEHASH,
                                     _msgSender(),
                                     totalCollateral,
                                     params.expiry,
keccak256(abi.encodePacked(params.anchorPrices)),
                                     params.collateralAtRisk,
                                     params.makerCollateral,
                                     params.deadline,
                                     address(this)))
        ));
>>      require(params.maker.isValidSignatureNow(digest,
params.makerSignature), "Vault: invalid maker signature");
        consumeSignature(params.makerSignature);
```

`isValidSignatureNow` first attempts to verify the digest using `ecrecover`. If that fails, it will make an ERC1271 call to the `params.maker` address. This creates a vulnerability where an attacker can frontrun the `mintProducts` call with their own parameters (`vault`, `totalCollateral`, `mintParams.maker`, etc.), while keeping `mintParams.makerSignature` and `signatures` unchanged. Since the parameters are manipulated, the `tryRecover` function will fail:

```
    function isValidSignatureNow(
        address signer,
        bytes32 hash,
        bytes memory signature
    ) internal view returns (bool) {
>>      (address recovered, ECDSAUpgradeable.RecoverError error) =
ECDSAUpgradeable.tryRecover(hash, signature);
        if (error == ECDSAUpgradeable.RecoverError.NoError && recovered
== signer) {
            return true;
        }

>>      (bool success, bytes memory result) = signer.staticcall(

abi.encodeWithSelector(IERC1271Upgradeable.isValidSignature.selector,
hash, signature)
        );
```

As a result, the ERC1271 fallback will be triggered, calling the attacker's maker address, which will pass successfully. This prevents the transaction from reverting, allowing the manipulated parameters to be transmitted.

Recommendation: It is recommended to validate `mintParams.maker` addresses in the `mintProducts` function.

SOFA.org: Fixed with @173c841

Zenith: Verified - Added a verification to ensure the `mintParams.maker` address is confirmed by an authorized maker.

[H-3] `vault` parameter is not verified in `burnProducts()`

Severity: High | Status: Resolved

Context:

- Automator.sol#L177

Description: When calling the `burnProducts` function, users are allowed to specify arbitrary `vault` addresses in the `products` argument, which can lead to the following exploits:

1. Vault inflation attack

- User A deposits 1 wei of collateral (WETH).
- They use a custom `vault` which will send 1 WETH to the automator in a `burnBatch` callback.
- `totalCollateral` = 1.000000000000000001. User B deposits 1 ETH but receives 0 shares.
- User A withdraws 2 ETH + 1 wei.

2. Re-entrancy attack An attacker could re-enter `burnProducts` in a custom vault and inflate profit values:

- The attacker calls `burnProducts` with their custom `vault` address, where the parameters don't matter.
- During the `burnBatch` callback, the attacker's vault calls `burnProducts` again, but this time with a legitimate product in the parameters.
- Let's say the position for this product is 1000 and the earned amount is also 1000, meaning there's no actual profit.
- This position is burned, and we return to the original `burnBatch`. From the Automator's point of view, `balanceAfter - balanceBefore = 1000`, so it calculates the profit as 1000 - 0 (the fake position), resulting in a profit of 1000.
- This way, even though no profits were received, the `totalCollateral` will still be inflated.

Recommendation: It is recommended to implement an allowlist for vault addresses in the `burnProducts` function. Alternatively, consider adding reentrancy guards to all user-facing functions and minting an initial amount of "dead" shares.

SOFA.org: Resolved with the following commits: @a4fdeb5, @22b726a, @0c389bc

Zenith: Verified. The reentrancy vulnerability has been mitigated by adding `nonReentrant` modifiers, and an initial share mint will be conducted to prevent vault inflation attacks.

[H-4] Products for each `_positions[id]` can be partially burnt in `burnProducts()` to distort PnL calculation

Severity: High	Status: Resolved

Products for each `_positions[id]` can be partially burnt in `burnProducts()` to distort PnL calculation

Context:

- Automator.sol#L177-L186

Description:

There are two issues with `burnProducts()` that will affect the PnL calculation.

The first issue is that the `id` for each iteration of the `burnProducts()` is obtained by using the index `0` of `products[i].products`, as follows:

```
    bytes32 id = keccak256(abi.encodePacked(products[i].vault,
products[i].products[0].expiry, products[i].products[0].anchorPrices));
```

This assumes that `products[i].products` all contains the same `expiry`and `anchorPrices`. However, that is not true, as it is possible to `burnBatch()` products with different `expiry` and `anchorPrices`. While the PnL calculation is based on the position using the index `0` product.

The second issue is due to the different product `id` format used in Automator as compared to the product vault.

For each iteration in `burnProducts()`, it retrieves the total collateral amount invested in the product `id` from `_positions[id]`. It will then perform `delete _positions[id]` to reset the amount invested in the product `id`.

These assume that all the products for `id` are burned at the same time in `burnBatch()`.

However, that is incorrect as it is possible for a partial amount of product `id` to be burned in `burnBatch()`.

That is because the product `id` for Automator is based on `(expiry, anchorPrices)`, while the SmartTrend vault product `id` are based on `(expiry, anchorPrices, collateralAtRiskPercentage, isMaker)`.

For example, if maker mint the following two products for the same SmartTrend vault,

- `(vault X, expiry = T1, anchorPrices = [x,y], collateralAtRiskPercentage = 5%)` with `10e18` collateral

- `(vault X, expiry = T1, anchorPrices = [x,y], collateralAtRiskPercentage = 10%)` with `10e18` collateral
- the total invested amount `_positions[id]` will be `20e18` collateral as it is only based on `(expiry = T1, anchorPrices = [x,y])`

However, anyone can just call `burnProducts()` for only one of the minted product That will use the total `20e18` invested collateral amount for PnL calculation, resulting in a loss. It will also set the invested amount invested amount `_positions[id]` to 0.

The next product burn will have a invested amount of 0 as it has been reset. That will result in a significant profit earned, resulting in a high fee paid on it.

```
    function burnProducts(
           ProductBurn[] calldata products
       ) external {
           uint256 totalEarned;
           uint256 totalPositions;
           uint256 fee;
           for (uint256 i = 0; i < products.length; i++) {
               uint256 balanceBefore = collateral.balanceOf(address(this));
               IVault(products[i].vault).burnBatch(products[i].products);
               uint256 balanceAfter = collateral.balanceOf(address(this));
               uint256 earned = balanceAfter - balanceBefore;
               totalEarned += earned;
>>>            bytes32 id = keccak256(abi.encodePacked(products[i].vault,
products[i].products[0].expiry, products[i].products[0].anchorPrices));
>>>            totalPositions += _positions[id];
               if (earned > _positions[id]) {
                   fee += (earned - _positions[id]) *
IFeeCollector(feeCollector).tradingFeeRate() / 1e18;
               }
>>>            delete _positions[id];
           }
           if (fee > 0) {
               totalFee += fee;
               totalEarned -= fee;
           }
           if (totalEarned > totalPositions) {
               totalCollateral += totalEarned - totalPositions;
           } else if (totalEarned < totalPositions) {
               totalCollateral -= totalPositions - totalEarned;
           }

           emit ProductsBurned(products, totalCollateral, fee);
       }
```

Recommendation:

The first issue can be resolved by using `IVault(products[i].vault).burn()` instead of `IVault(products[i].vault).burnBatch()` so that each of the product burn has its own PnL calculation.

The second issue can be resolved by using the same product id format as the product vault to track the invested collateral.

SOFA.org: Fixed in both @200451f and @fe1b392.

Zenith: Verified

4.2 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] Users can frontrun losses after 7 days of depositing by requesting redemption upon deposit

Severity: Medium	Status: Resolved

Context:

- Automator.sol#L127-L143

Description:

The timelock on redemption is designed to prevent users from arbitraging by sandwiching profit from `burnProducts()` with deposit before and withdrawal after.

However, the timelock does not handle the case where users can frontrun heavy losses from `burnProducts()`.

That is because users can trigger `withdraw()` upon deposit and leave it as a pending redemption, allowing them to withdraw anytime after 7 days. When a heavy loss occur, they can choose to withdraw and frontrun `burnProducts()` to evade the loss.

```
    function claimRedemptions() external {
        require(_redemptions[_msgSender()].pendingRedemption > 0,
"Automator: no pending redemption");
        require(block.timestamp >=
_redemptions[_msgSender()].redemptionRequestTimestamp + 7 days,
"Automator: early redemption");

        uint256 pendingRedemption =
_redemptions[_msgSender()].pendingRedemption;
        uint256 amount = pendingRedemption * getPricePerShare() / 1e18;
        require(collateral.balanceOf(address(this)) >= amount,
"Automator: insufficient collateral to redeem");

        totalPendingRedemptions -= pendingRedemption;
        _redemptions[_msgSender()].pendingRedemption = 0;
        totalCollateral -= amount;

        _burn(_msgSender(), pendingRedemption);
        collateral.safeTransfer(_msgSender(), amount);
        emit RedemptionsClaimed(_msgSender(), amount, pendingRedemption);
    }
```

Recommendation:

One solution is to impose a withdrawal period, to only provide a small time window for `claimRedemptions()` such as 1 day. That will makes it harder for users to try to frontrun any losses.

When the grace period is over, the pending redemption is cancelled and user has to re-request again.

```
    function claimRedemptions() external {
        require(_redemptions[_msgSender()].pendingRedemption > 0,
"Automator: no pending redemption");
        require(block.timestamp >=
_redemptions[_msgSender()].redemptionRequestTimestamp + 7 days,
"Automator: early redemption");


+       if(block.timestamp >=
_redemptions[_msgSender()].redemptionRequestTimestamp + 7 days + 1 days)
{
+            // past 1 day withdrawal period
+            totalPendingRedemptions -= pendingRedemption;
+           _redemptions[_msgSender()].pendingRedemption = 0;
+            return;
+        }


        uint256 pendingRedemption =
_redemptions[_msgSender()].pendingRedemption;
        uint256 amount = pendingRedemption * getPricePerShare() / 1e18;
        require(collateral.balanceOf(address(this)) >= amount,
"Automator: insufficient collateral to redeem");

        totalPendingRedemptions -= pendingRedemption;
        _redemptions[_msgSender()].pendingRedemption = 0;
        totalCollateral -= amount;

        _burn(_msgSender(), pendingRedemption);
        collateral.safeTransfer(_msgSender(), amount);

        emit RedemptionsClaimed(_msgSender(), amount, pendingRedemption);
    }
```

SOFA.org: Fixed with @ec2b243. We extended the window period to 3 days. I adjusted the proposed changes to simplify user operations. Additional fix for `totalPendingRedemptions` with @819f840. Also removed restriction to allow users to change redemption amount in `withdraw()` in @d90325c.

Zenith: Verified—The issue is resolved by reverting `claimRedemptions()` when the 3-day withdrawal window has passed.

[M-2] Automator will be bricked when `totalCollateral` reaches zero

Severity: Medium	Status: Acknowledged

Context:

- Automator.sol#L110

Description:

The function `deposit()` calculates the amount of share to mint based on the formula `shares = amount * totalSupply() / totalCollateral`.

However, this will always revert if `totalCollateral` reduces to zero due to complete losses for all minted products. The issue will cause the Automator to be bricked and unable to accept any new deposits, without any ways to recover the Automator from it.

```
    function deposit(uint256 amount) external {
        collateral.safeTransferFrom(_msgSender(), address(this), amount);
        uint256 shares;
        if (totalSupply() == 0) {
            shares = amount;
        } else {
            //@audit this will always revert when `totalCollateral == 0`
>>>         shares = amount * totalSupply() / totalCollateral;
        }
        totalCollateral += amount;
        _mint(_msgSender(), shares);
        emit Deposited(_msgSender(), amount, shares);
    }
```

Recommendation:

This can be resolved with any of the following,

1. Have a process to re-deploy the vault and deprecate the bricked vault (recommended). OR
2. Implment a reset function to burn all the shares of current users to reset `totalSupply()` to 0.

SOFA.org: Acknowledged, if it happens, we will choose the first plan.

Zenith: Option 1 will be adopted - re-deploy vault and deprecate brick vault when issue occurs.

4.3 Low Risk

A total of 1 low risk findings were identified.

[L-1] Harvesting fees can prevent depositors from claiming their collateral

Severity: Low Status: Resolved

Context:

- Automator.sol#L201-L208

Description: Automator depositors can withdraw their collateral by creating a pending request. After seven days, the request can be executed, resulting in shares being burned and collateral transferred to the owner:

```
    function claimRedemptions() external {
        require(_redemptions[_msgSender()].pendingRedemption > 0,
"Automator: no pending redemption");
        require(block.timestamp >=
_redemptions[_msgSender()].redemptionRequestTimestamp + 7 days,
"Automator: early redemption");

        uint256 pendingRedemption =
_redemptions[_msgSender()].pendingRedemption;
        uint256 amount = pendingRedemption * getPricePerShare() / 1e18;
        require(collateral.balanceOf(address(this)) >= amount,
"Automator: insufficient collateral to redeem");

        totalPendingRedemptions -= pendingRedemption;
        _redemptions[_msgSender()].pendingRedemption = 0;
        totalCollateral -= amount;

        _burn(_msgSender(), pendingRedemption);
        collateral.safeTransfer(_msgSender(), amount);

        emit RedemptionsClaimed(_msgSender(), amount, pendingRedemption);
    }
```

At the same time, makers can send funds from the Automator to vaults, provided that the collateral in the Automator is sufficient to cover existing pending redemptions:

```
    function mintProducts(
        ProductMint[] calldata products,
        bytes calldata signature
    ) external {
        ---SNIP---

>>      require(collateral.balanceOf(address(this)) >=
totalPendingRedemptions * getPricePerShare() / 1e18, "Automator: no
enough collateral to redeem");
    }
```

However, this check is missing in the `harvest` function, where the owner can transfer collected fees to the receiver. Consider the following scenario:

- The current state of the Automator shows that 10 WETH have been deposited, and 1 WETH has been received as fees.
- A user decides to withdraw 5 WETH.
- Meanwhile, 6 WETH are sent to the vault, leaving a total of 5 WETH in the contract.
- The owner calls `harvest`, transferring 1 WETH, which leaves only 4 WETH in the contract.
- When the claim time comes, the user won't be able to withdraw their 5 WETH due to insufficient funds.

Recommendation: It is recommended to verify that harvested fees do not leave the Automator in a state where all pending redemptions cannot be claimed.

SOFA.org: Fixed in the following commit - @423d25b

Zenith: Verified - Added a check to ensure sufficient funds are available for executing pending redemptions and harvesting fees.