Pareto Security Review

Security Review

21.04.2025

Conducted by :

Pyro, Security Researcher

samuraii77, Security Researcher

Table of Contents

1 Disclaimer

Audits are a time, resource, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

2 System overview

USP is a synthetic dollar that is backed primarily by Pareto Credit Vaults. Pareto Credit Vaults are a collection of institutional-grade lending strategies that generate yield on the underlying assets via proprietary strategies of institutional borrowers.

3 Executive summary

3.1 Overview

Field	Value
Project Name	Pareto
Repository	https://github.com/pareto-credit/USP
Commit hash	ea506345ef2ba67eb865458ceb3d79f9e33c41c1
Remediation	ea295f0a712ac23eb02308f05d8891850db938af
Methods	Manual review

3.2 Timeline

Version	Date	Description
v0.1	16.04.2025	Audit kick-off
v0.1	21.04.2025	Preliminary report
v1.0	21.04.2025	Mitigation review

3.3 Scope

Contracts
src/Constants.sol
src/EmergencyUtils.sol
src/ParetoDollar.sol
src/ParetoDollarQueue.sol
src/ParetoDollarStaking.sol
src/interfaces/IParetoDollar.sol
src/interfaces/IParetoDollarQueue.sol

3.4 Issues Found

Severity	Count
High	1
Medium	3
Low	5

4 Findings

4.1 High Severity

[H-01] Storage will be broken if 2 yield sources share the same underlying token upon yield source removal

Removing a yield source works by utilizing the swap and pop algorithm:

```
    for (uint256 i = 0; i < sourcesLen; i++) {
      if (address(_sources[i].token) == address(_ys.token)) {
        allYieldSources[i] = _sources[sourcesLen - 1];
        allYieldSources.pop();
        break;
      }
    }
```

However, the token equality we are searching for is not ideal as if 2 yield sources share the same underlying token (which is extremely likely), then this might ruin the storage by popping an incorrect yield source. Imagine the following scenario:

1. There are 2 yield sources (`A` and `B`) with `USDC` as an underlying token, they are in the array as follows `[A, random_yield_source, B]`

2. We want to remove the `B` yield source

3. In the loop we are running, we find an equality between `USDC` and the underlying token of yield source `A`, thus we apply the swap and pop method on it and the yield source array is now `[B, random_yield_source]` while the mapping for `B` is deleted which is incorrect and will cause all sorts of different issues as the storage will now be completely destroyed

Storage for the 2 sources (one to remove and the one actually removed from the array) is completely broken. This causes almost all functions to work incorrectly and return incorrect results.

Recommendation:

Consider searching for equality based on the `source` field instead which is guaranteed to be unique due to this check when adding a yield source:

```
    if (address(yieldSources[_source].token) != address(0)) {
      revert YieldSourceInvalid();
    }
```

Resolution: Fixed

4.2 Medium Severity

[M-01] `scaledNAVERC4626()` will often inflate the asset value

`scaledNAVERC4626()` is as follows:

```
function scaledNAVERC4626(IERC4626 vault) internal view returns (
    uint256) {
  // ERC4626 vault
  // convertToAssets returns value in underlying decimals so we
     scale it to 18 decimals
  return vault.convertToAssets(vault.balanceOf(address(this))) *
     10 ** (18 - IERC20Metadata(vault.asset()).decimals());
}
```

It works by simply getting the shares we have, converting them to assets and turning them into 18 decimals. However, according to the ERC4626 specification, `convertToAssets()` does not include things such as fees, variations depending on the caller, slippage, etc. Directly quoting the actual behaviour of the function: > This calculation MAY NOT reflect the “per-user’s” price-per-share, and instead should reflect the “average-user’s” price-per-share, meaning what the average user should expect to see when exchanging to and from.

This causes our asset value to be artifically inflated, for example:

1. The yield source vault has a redemption fee of 10%, we have 100 shares, at a 1:1 share to asset ratio

2. `scaledNAVERC4626()` will return 100 assets (ignoring decimals) due to how all ERC4626 vaults are supposed to

3. The actual value we have control over is 90 due to the fee which is applied upon a redemption

`scaledNAVERC4626()` returns an artificially inflated value which causes functions to work incorrectly, e.g. minting gain to the staking contract depositors when our collateral value is actually lower than the supply:

Recommendation:

Consider using `previewRedeem()` instead which mocks the actual result of a potential `redeem()` call. Note that this will cause the max cap checks to be slightly incorrect (as we

won't exactly get the deposited assets but instead our redemption value), however this is not an issue as the cap can be changed, the difference will be small and there is practically no impact even if we go slightly above the cap.

Resolution: Fixed

[M-02] The same validity period across all assets is problematic

The oracle validity period is the same for all different collateral assets in `ParetoDollar`:

```
oracleValidityPeriod = 24 hours;
```

Then, upon getting prices for an asset, we use it as follows:

```
    if (answer > 0 && (_oracleValidity == 0 || (updatedAt >= block.
       timestamp - _oracleValidity))) {
      ...
    }
```

Using the same validity period is problematic as different feeds have different staleness periods. For example:

1. `AssetA` has a staleness period of 1 hour, `AssetB` has a staleness period of 24 hours
2. Everything will work correctly when fetching prices for `AssetB`, however when fetching prices for `AssetA`, an extremely stale price (e.g. 23 hours old) will still be considered fresh

Recommendation:

Consider adding a validity period field to the `CollateralInfo` struct and use it instead

Resolution: Fixed

[M-03] Removing a collateral may indirectly lower the protocol's yield

When removing a collateral we re-order the rest, by changing the one at the last place to the place of the removed one:

```
        for (uint256 i = 0; i < collateralsLen; i++) {
            if (_collaterals[i] == token) {
                collaterals[i] = _collaterals[collateralsLen - 1];
                collaterals.pop();
                break;
            }
        }
```

However that will cause them to change order, and this order is used when fulfilling claims with `claimRedeemRequest`. Where the first collateral is drained first, and if the funds are not enough we move to the second and so on.

```
        address[] memory allCollaterals = par.getCollaterals();
        uint256 collateralsLen = allCollaterals.length;

        // loop through the collaterals and transfer them to the
           user up to the USP amount requested
        // we treat collaterals (USDC, USDT) at 1:1 with USP
        for (uint256 i = 0; i < collateralsLen && _amountLeft > 0; i
           ++) {
            _collateralToken = IERC20Metadata(allCollaterals[i]);
```

However some collaterals, may support higher interest rates at the time, and if the accidentally end up first in the queue then the vault will have less of them. This will lower the APY.

For example, imagine there are 3 tokens, `[A, B, C]` with APY of `[5%, 5%, 15%]` respectively where currently token A is the one which first takes care of the withdrawals (due to it being first in the array). If token A is removed, the array will be reordered as `[C, B]` due to the swap and pop algorithm. Now, when a user withdraws, the first token balance that we will try to send will be token C which has the highest yield. Lowering the highest APY deposits will cause less yield in the future.

Recommendation:

Consider having a custom queue just for withdraws. This way if currently USDT gives the lowest yield we can pay claims in it, and next month if USDC starts to give the lowest yield we can switch USDC to be the first to be withdraw.

Resolution: Acknowledged

4.3 Low Severity

[L-01] ParetoDollarStaking is not ERC4626 compliant

The following 4 functions are not compliant with the ERC4626 specification: - `maxDeposit()` - `maxMint()` - `maxWithdraw()` - `maxRedeem()`

They must factor in global and user specific limits: >MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0.

A limit that we have to take into account but we do not is the fact that it is pausable and it will cause all deposits and withdrawals to revert:

```
function _withdraw(address caller, address receiver, address
    _owner, uint256 assets, uint256 shares) internal override {
  _requireNotPaused();
  super._withdraw(caller, receiver, _owner, assets, shares);
}
```

Currently, the functions listed above are not overridden and they just return their default values, e.g. `type(uint256).max` for `maxDeposit()` which would be incorrect when the contract is paused and will cause issues for all integrators relying on correct data from the listed functions.

Recommendation:

Override the listed functions and return 0 if we are in a paused state.

Resolution: Fixed

[L-02] Constants will not work on all chains besides ETH

The current system uses a bunch of constants for addresses, used when initializing.

```
    function initialize(...) public initializer {
        // ...
        IERC20Metadata(USDS).safeIncreaseAllowance(USDS_USDC_PSM,
           type(uint256).max);
    }
```

However these addresses will change depending on the network, meaning that the current system will work only on ETH.

Recommendation:

Avoid using constant for addresses (as they change depending on the network). Consider having those variables in the inputs, this way contracts can be deployed and used on other chains too.

Resolution: Partially fixed

[L-03] No min/max price checks

There is no min/max checks. While min/max prices have been deprecated, there are still some feeds with them.

```
    function _getScaledOracleAnswer(address oracle, uint8
       feedDecimals) internal view returns (uint256) {
        (, int256 answer,, uint256 updatedAt,) = IPriceFeed(oracle).
           latestRoundData();
        uint256 _oracleValidity = oracleValidityPeriod;

        if (answer > 0 && (_oracleValidity == 0 || (updatedAt >=
           block.timestamp - _oracleValidity))) {
            return uint256(answer) * 10 ** (18 - feedDecimals);
        }

        return 0;
    }
```

Incorrect price might get accepted as correct, and allow use of dangerous (at that time) collateral.

Recommendation:

Consider adding the min/max price check, however the issue could also be acknowledged as the current 3 feeds don't require them and the min/max prices are almost fully deprecated.

Resolution: Acknowledged

[L-04] Changing the fee in ParetoDollarStaking does not handle the pending yield

Updating fee params works as follows:

```
  function updateFeeParams(uint256 _fee, address _feeReceiver)
     external {
    _checkOwner();
    if (_fee > MAX_FEE) {
      revert FeeTooHigh();
    }
    fee = _fee;
    feeReceiver = _feeReceiver;
  }
```

The fee is applied when rewards are deposited into the staking contract, e.g. 10% of the total rewards sent are sent to the fee receiver while the other 90% are left for the stakers. The issue is that updating the fee does not handle the pending yield, potentially resulting in the following scenario:

1. Fee is currently 10%, collateral value is 100 and the total supply of USP is 90, thus 10 USP should be sent to the staking contract of which 1 should be sent to the fee receiver and 9 should be vested to stakers

2. Fee is changed to 20% which does not handle the pending yield of 10 USP

3. When `depositYield()` is called, users will only be vested 8 of the 10 tokens instead of the 9 they expected

Recommendation:

The below diff can be applied so pending yield is vested based on the current fee. Note that the access control in `distributeYield()` must be changed to allow the call.

```
  function updateFeeParams(uint256 _fee, address _feeReceiver)
     external {
    ...
+   IParetoDollarQueue(queue).depositYield();
    fee = _fee;
    feeReceiver = _feeReceiver;
  }
```

Resolution: Acknowledged

[L-05] Users might receive less assets/shares than anticipated

Upon depositing/withdrawing in the staking contract, there is no slippage.

In both scenarios, there is a way for users to receive less than anticipated: - deposit case - when a user deposits and the transaction executes during a vest (he also might have submitted it beforehand), the asset value will have gone up, resulting in less shares minted (as his asset deposit is a lower % of the total assets) - withdraw case - the protocol has a mechanism to absorb borrower losses by taking out assets from the staking contract and then burning them, causing the share value to go down. If a user's transaction is executed after such an event (he might have submitted the transaction prior to that), he will receive less assets than what he had anticipated

Recommendation:

Consider adding wrapper functions which accept slippage as an input.

Resolution: Acknowledged