Ubiquity DAO Security Review

Conducted by: 0xadrii

April 18, 2024 - April 20, 2024

Contents

1. Introduction

1.1 About 0xadrii

I specialize in conducting smart contract audits as an independent security researcher. My expertise includes a proven track record in public audit contests with numerous top 3 finishes and bug bounties, along with extensive experience in evaluating complex and high-profile protocols. You can find my previous work **here** or reach out on Twitter at **@0xadrii**.

1.2 About Ubiquity DAO

Ubiquity DAO aspires to be the bank of the metaverse. The Ubiquity Dollar serves as the flagship product towards that end. Their vision includes providing a stablecoin for the DAO economy and to incorporate reliable, seamless, and effective economic integration into platforms and products amplifying the Ubiquity Dollar's reach and utility beyond just the DeFi sandbox.

1.3 Disclaimer

This report presents an analysis conducted within specific parameters and timeframe, relying on provided materials and documentation. It **does not** encompass all possible vulnerabilities and should not be considered exhaustive. The review and accompanying report are provided on an 'as-is' and 'as-available' basis, without any express or implied warranties. Additionally, this report does not endorse any particular project or team, nor does it guarantee the absolute security of the project.

2. Risk classification

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

2.1 Impact

- High: Funds are **directly** at risk, or a **severe** disruption of the protocol’s core functionality.
- Medium: Funds are **indirectly** at risk, or **some** disruption of the protocol’s functionality.
- Low: Funds are **not** at risk.

2.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized.
- Medium - only conditionally possible or incentivized, but still relatively likely.
- Low - really improbable, requires little-to-no incentive.

2.3 Action required for severity levels

- High: must fix as soon as possible.
- Medium: should fix.
- Low: Could fix.

3 Executive summary

Over the course of two days in total, Ubiquity DAO engaged with 0xadrii to review Ubiquity Dollar. In this period of time, a total of 11 issues were found.

3.1 Overview

Project	**Ubiquity Dollar**
Repository	https://github.com/ubiquity/ubiquity-dollar
Audit timeline	April 18th, 2024 - April 20th, 2024

3.2 Issues found

Severity	**# of issues**
High	0
Medium	2
Low	1
Gas optimization	6
Informational	2
Centralization	0
Total issues	11

4 Findings summary

ID	Description	Status
M-01	Pairing Dollar with 3CRV will still make Dollar follow a price of $1.03, even if StableSwapMetaNG pools are used	Fixed in PR #933
M-02	Adding collateral enabled checks for redemption collections might make users uncapable of redeming their funds	Acknowledged
L-01	Critical protocol variables should be capped	Fixed in PR #932
INFO-01	Missing event emission after setting role admin	Acknowledged
GAS-01	Unnecessary msg.data.length check	Acknowledged
GAS-02	Using SafeMath library to perform calculations is unnecessary	Acknowledged
GAS-03	Access to ubiquityPoolStorage().collateralRatio can be cached	Acknowledged
GAS-04	Logic to check if a collateral exists can be optimized	Acknowledged
GAS-05	Avoid unnecessary empty initializations	Acknowledged
GAS-06	Use unchecked for operations not expected to overflow	Acknowledged
CENTR-01	Setting the collateral ratio should be timelocked	Acknowledged

5 Findings

High

None.

Medium

[M-01] - Pairing Dollar with 3CRV will still make Dollar follow a price of $1.03, even if StableSwapMetaNG pools are used

Context: LibUbiquityPool.sol#L368

Severity: Medium

Every time a mint/redeem takes place, it is first validated that the dollar price is between the `mintPriceThreshold` / `redeemPriceThreshold` , which is done to prevent supply modifications that would impact Dollar's peg.

In order to check the price threshold, a custom Dollar/3CRV Curve StableSwapMetaNG pool is used, from which the price is obtained by triggering the pool's `price_oracle` function:

```
function getDollarPriceUsd()
        internal
        view
        returns (uint256 dollarPriceUsd)
    {
        // load storage shared across all libraries
        AppStorage storage store = LibAppStorage.appStorage();
        // get Dollar price from Curve Metapool (18 decimals)
        uint256 dollarPriceUsdD18 = ICurveStableSwapMetaNG(
            store.stableSwapMetaPoolAddress
        ).price_oracle(0);
        // convert to 6 decimals
        dollarPriceUsd = dollarPriceUsdD18
            .mul(UBIQUITY_POOL_PRICE_PRECISION)
            .div(1e18);
    }
```

The problem with this approach is that 3CRV price is not actually stable at $1, and rather fluctuates (at the time of writing, 3CRV price is $1.03 approximately). This price actually affects the StableSwapMetaNG computations because the metapool's virtual price is fetched and used as the metapool's rate in all the pool's calculations. As shown in the following snippet, the oracle update will be performed considering the new `xp` value, which has been computed via the pool's `rates` and balances:

```
# CurveStableSwapMetaNG.vy

@view
@internal
def _stored_rates() -> uint256[N_COINS]:
    """
    @notice Gets rate multipliers for each coin.
    @dev If the coin has a rate oracle that has been properly initialis
         this method queries that rate by static-calling an external
         contract.
    """
    rates: uint256[N_COINS] = [rate_multiplier, StableSwap(BASE_POOL).g

    ...


@external
@nonreentrant('lock')
def add_liquidity(
    _amounts: uint256[N_COINS],
    _min_mint_amount: uint256,
    _receiver: address = msg.sender
) -> uint256:

      ...

    rates: uint256[N_COINS] = self._stored_rates()

    ...

    if total_supply > 0:

          ...

        xp: uint256[N_COINS] = self._xp_mem(rates, new_balances)
        D1 = math.get_D([xp[0], xp[1]], amp, N_COINS)  # <------- Reuse 
        # we do unsafe div here because we already did several safedivs
        mint_amount = unsafe_div(total_supply * (D1 - D0), D0)
        self.upkeep_oracles(xp, amp, D1)
```

This will make the pool tend to follow a price where balance is considered when the pool ratio sits at 0.97:1 (Dollar:3CRV), making the price of Dollar always tend to be smaller than $1 on equilibrium.

Recommendation

It is recommended to pair Dollar with an actual stable coin, instead of 3CRV. The regular **CurveStableSwapNG** pool ***can be used instead of* CurveStableSwapMetaNG.**

[M-02] - Adding collateral enabled checks for redemption collections might make users uncapable of redeming their funds

Context: LibUbiquityPool.sol#L714

Severity: Medium

The protocol team implemented a fix for the bug described in this Sherlock issue. The fix describes how the `collateralEnabled` check should be added in the `collectRedemption` function as well to ensure users can't collect redemptions when collateral is disabled. However, this recommendation is wrong, given that users should always be capable of withdrawing their queued funds, as collecting is a process performed after redeeming, which is the actual flow that impacts Dollar's peg. Adding the `collateralEnabled` check is problematic in the sense that if Ubiquity team decides to disable a collateral in the future, some redemptions might be already queued, so users won't be able to withdraw their funds until the collateral is enabled again.

Recommendation

Ideally, both `collateralEnabled` and `isRedeemPaused` checks should be removed from the `collectRedemption` function, given that this will prevent funds from being temporarily locked in the contract.

Low

[L-01] - Critical protocol variables should be capped

Context: LibUbiquityPool.sol#L1040**,** LibUbiquityPool.sol#L1009, LibUbiquityPool.sol#L1115

Severity: Low

The code contains several critical storage variables that can be directly set by governance and which affect critical flows. Some of these variables include:

- `mintingFee/redemptionFee`
- `collateralRatio`
- `redemptionDelayBlocks`

All of these variables should be capped with reasonable maximum values in their corresponding setter functions, so that in the event of compromised keys/mistakes they can’t be set to values that could lead to protocol malfunction.

Recommendation

Add checks in the `setFees`, `setCollateralRatio` and `setRedemptionDelayBlocks` that force such variables to be set between reasonable values.

Informational

[INFO-01] - Missing event emission after setting role admin

Context: AccessControlInternal.sol#L91

Severity: Informational

PR #880 incorporates a function to set an admin role for a specific role. However, this new setter function misses emitting an event with the corresponding change. Events are relevant to be emitted whenever significant parameters of a contract are modified, or whenever relevant interactions are performed, enabling seamless communication with clients regarding state changes and facilitating application subscriptions to these updates.

Recommendation:

Consider emitting an event in the `_setRoleAdmin` function.

Gas

[GAS-01] - Unnecessary `msg.data.length` check

Context: Diamond.sol#L53

Severity: Gas

PR#912 fixes a Sherlock issue where it is stated that attackers can bypass Diamond whitelisting functions. From the Sherlock issue, the following scenario could arise:

1. Admins sets a relevant function such as `evaSafesFactory()` in FACET1 as whitelisted function, with a signature of `0x12d05b00`.
2. The attacker is then able to call the facet with `data = 0x12d05b`.
3. `fallback()` function in the diamond proxy would use `msg.sig` to find the facet related to the function signature.

4. Solidity would pad the `0x12d05b` with zeros and `msg.sig = 0x12d05b00` and code would validate FACET1 as target facet.
5. Then code would delegate call FACET1 address with `data = 0x12d05b` and in that target address's code the fallback would be executed.

The example assumes that the function is always callable, however it overlooks the fact that facets already implement access control by themselves, as seen for example in the `[setCollateralChainLinkPriceFeed` function in the UbiquityPoolFacet](https://github.com/ubiquity/ubiquity-dollar/blob/development/packages/contracts/src/dollar/facets/UbiquityPoolFacet.sol#L228). This makes the length check unnecessary, given that even if an admin function is called by a user, it will be properly protected in the corresponding facet.

Recommendation

The `msg.data.length` check is unnecessary and can be removed for gas optimizing purposes.

[GAS-02] - Using SafeMath library to perform calculations is unnecessary

Context: General

Severity: Gas

Currently, all math operations are performed using the SafeMath library, which adds extra security checks to prevent critical security issues, such as overflows/underflows. However, since Solidity 0.8.0, these type of security checks are built-in Solidity, so SafeMath is no longer needde.

Recommendation

Remove the SafeMath library to dramatically reduce gas costs.

[GAS-03] - Access to `ubiquityPoolStorage().collateralRatio` can be cached

Context: LibUbiquityPool.sol#L518

Severity: Gas

When minting, `ubiquityPoolStorage().collateralRatio` is frequently accessed due to the newly-incorporated fractional logic. Accessing to this storage variable can be cached instead of frequently accessed to reduce gas consumption.

Recommendation

Cache `ubiquityPoolStorage().collateralRatio` in `mintDollar` in the following way:

```
function mintDollar(
        uint256 collateralIndex,
        uint256 dollarAmount,
        uint256 dollarOutMin,
        uint256 maxCollateralIn,
        uint256 maxGovernanceIn,
        bool isOneToOne
    ) {
    ....

+       uint256 currentCollateralRatio = ubiquityPoolStorage().collatera
             if (
            isOneToOne ||
-            ubiquityPoolStorage().collateralRatio >=
+           currentCollateralRatio >=
            UBIQUITY_POOL_PRICE_PRECISION
        ) {
            // get amount of collateral for minting Dollars
            collateralNeeded = getDollarInCollateral(
                collateralIndex,
                dollarAmount
            );
            governanceNeeded = 0;
-        } else if (ubiquityPoolStorage().collateralRatio == 0) {
+        } else if (currentCollateralRatio == 0) {
            // collateral ratio is 0%, Dollar tokens can be minted by p
            collateralNeeded = 0;
            governanceNeeded = dollarAmount
                .mul(UBIQUITY_POOL_PRICE_PRECISION)
                .div(getGovernancePriceUsd());
        } else {

            // fractional, user has to provide both collateral and Gove
            uint256 dollarForCollateral = dollarAmount
-                .mul(ubiquityPoolStorage().collateralRatio)
+                .mul(currentCollateralRatio)
                .div(UBIQUITY_POOL_PRICE_PRECISION);

            uint256 dollarForGovernance = dollarAmount.sub(dollarForCol

            collateralNeeded = getDollarInCollateral(
                collateralIndex,
                dollarForCollateral
            );
            governanceNeeded = dollarForGovernance
                .mul(UBIQUITY_POOL_PRICE_PRECISION)
                .div(getGovernancePriceUsd());
        }
```

[GAS-04] - Logic to check if a collateral exists can be optimized

Context: LibUbiquityPool.sol#L234

Severity: Gas

Currently, the `collateralExists` will verify if a collateral exists by iterating over all the collateral addresses. If the address is found, it means that the collateral exists:

```
function collateralExists(
        address collateralAddress
    ) internal view returns (bool) {
        UbiquityPoolStorage storage poolStorage = ubiquityPoolStorage()
        address[] memory collateralAddresses = poolStorage.collateralAd

        for (uint256 i = 0; i < collateralAddresses.length; i++) {
            if (collateralAddresses[i] == collateralAddress) {
                return true;
            }
        }
        return false;
    }
```

However, this method is extremely gas-intensive, and can be performed in a different way to decrement gas consumption.

Recommendation

A way to decrease gas consumption is to change the mapping in `isCollateralEnabled` from `bool` to `uint256`. Instead of using booleans to see if a collateral is enabled, uint256 can be used:

- 0: collateral does not exist and is not enabled
- 1: collateral exists but is not enabled
- 2: collateral exists and is enabled

This way, the `collateralExists` function can simply check if the `isCollateralEnabled` returns a number different from 0 to verify that a collateral actually exists.

[GAS-05] - Avoid unnecessary empty initializations

Context: General

Severity: Gas

The code initializes several variables to 0. This can be avoided, given that variables in solidity are initialized to 0/empty by default. This type of initializations can be seen for example in loops (initializing iterators to 0) or in the minting/redeeming fractional flow, where the `governanceNeeded` and `collateralNeeded` will be unnecesarily set to 0 depending on the situation:

```
    function mintDollar(
        uint256 collateralIndex,
        uint256 dollarAmount,
        uint256 dollarOutMin,
        uint256 maxCollateralIn,
        uint256 maxGovernanceIn,
        bool isOneToOne
    ) {
        ...

       if (
            isOneToOne ||
            ubiquityPoolStorage().collateralRatio >=
            UBIQUITY_POOL_PRICE_PRECISION
        ) {
            // get amount of collateral for minting Dollars
            collateralNeeded = getDollarInCollateral(
                collateralIndex,
                dollarAmount
            );
            governanceNeeded = 0;
        } else if (ubiquityPoolStorage().collateralRatio == 0) {
            // collateral ratio is 0%, Dollar tokens can be minted by p
            collateralNeeded = 0;
            governanceNeeded = dollarAmount
                .mul(UBIQUITY_POOL_PRICE_PRECISION)
                .div(getGovernancePriceUsd());
        } else {

            // fractional, user has to provide both collateral and Gove
            uint256 dollarForCollateral = dollarAmount
                .mul(ubiquityPoolStorage().collateralRatio)
                .div(UBIQUITY_POOL_PRICE_PRECISION);

            uint256 dollarForGovernance = dollarAmount.sub(dollarForCol

            collateralNeeded = getDollarInCollateral(
                collateralIndex,
                dollarForCollateral
            );
            governanceNeeded = dollarForGovernance
                .mul(UBIQUITY_POOL_PRICE_PRECISION)
                .div(getGovernancePriceUsd());
        }
```

Recommendation

Remove redundant 0 initializations.

[GAS-06] - Use `unchecked` for operations not expected to overflow

Context: General

Severity: Gas

If it is not possible for an operation to overflow, such operation should be wrapped in `unchecked {}` to save the gas that would have been used to check for an overflow. This can be useful for example in loops, where incrementing `i` can be done in an unchecked loop.

Recommendation

Wrap operations that can't overflow in `unchecked` blocks, such as:

```
function collateralExists(
        address collateralAddress
    ) internal view returns (bool) {
        UbiquityPoolStorage storage poolStorage = ubiquityPoolStorage()
        address[] memory collateralAddresses = poolStorage.collateralAdd

-       for (uint256 i = 0; i < collateralAddresses.length; i++) {
+       for (uint256 i = 0; i < collateralAddresses.length;) {
            if (collateralAddresses[i] == collateralAddress) {
                return true;
            }
+           unchecked {
+               ++i;
+            }
        }
        return false;
    }
```

Centralization

[CENTR-01] - Setting the collateral ratio should be timelocked

Context: LibUbiquityPool.sol#L1009

Severity: Gas

Ubiquity Dollar has adopted Frax's fractional mechanism, where the flow of minting and redeeming of tokens depends on the Collateral Ratio currently set in the protocol. While Frax's Collateral Ratio autorregulates via the `[refreshCollateralRatio](https://github.com/FraxFinance/frax-solidity/blob/96700` function, Ubiquity incorporates a setter function (`setCollateralRatio`) that allows governance to directly set the Collateral Ratio. In a situation of keys compromised, setting the Collateral Ratio can completely break the protocol mechanisms and break Dollar’s Peg. Additionally, several other setter functions might affect critical variables in the protocol.

Recommendation

Implement a Timelock that prevents critical protocol functions from being directly triggered.