Chainlink Uniswap Anchored View Audit

Smart Contract Security Assessment

Nov 29, 2021

ABSTRACT

Dedaub was commissioned to perform a security audit on an updated version of Compound's Uniswap Anchored View smart contract on branch feature/14246-uni-v3-native-twap at commit hash c2bd41df808f49ea5a3bd7abc78d8d5745a28b3d. The additional code can be found in this pull request. The audit also examined any other functionality highly related to UniswapAnchoredView.sol contract. Specifically, contracts UniswapConfig.sol and UniswapLib.sol. Two auditors worked over the codebase for a week.

SETTING & CAVEATS

The Uniswap Anchored View (UAV) is an oracle employed as part of the Compound Open Price Feed architecture and also leveraged by other protocols (e.g., Yearn). The UAV receives prices from Chainlink, but does not let them deviate from the Uniswap time-weighted average price (TWAP) by more than a certain factor (currently 15%). Also, it has two special cases:

- Under the owner's direction, the price of an asset is taken from the Uniswap TWAP alone;
- some tokens (USDC, USDT, TUSD) are considered to be 1-1 with the US dollar. Others (SAI) are considered to be at a fixed ratio with ETH. No authority can change this equivalence, even if Chainlink and Uniswap were to agree otherwise.

The functionality we audited concerns the update of the UAV for use with Uniswap v3, leveraging the new Uniswap v3 TWAP functionality.

The audit considered all layers, from protocol-level security threats down to math calculations. We spent significant time verifying correct scaling and Uniswap math. However, one should be aware that functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing. Since the project has an extensive test suite (which we did not audit) we expect that this angle is well-covered.

Similarly, we trust that any documented environmental setting is configured correctly. This, for instance, includes all parameters in the configuration (e.g., "reporterMultiplier" for every token), as well as the assumptions stated in the constructor code ("*the system*

must run for at least a single anchorPeriod before using; ... this contract will not be voted in by governance until prices have been updated through `validate`").

In addition to the above, the deployer should be aware that, **compared to the original UAV, the setting up of the contract for Uniswap v3 needs to include the expansion of the `observations` array of the Uniswap pool to enough positions to accommodate the pool's transactions that have occurred in the last `anchorPeriod`**. This requires calling `increaseObservationCardinalityNext()` on the pool. For instance, the BAT pool currently contains only a single observation, therefore attempts at computing TWAP will revert if there has been any BAT swap more recently than `anchorPeriod` seconds ago.

PROTOCOL-LEVEL CONSIDERATIONS

In inspecting the code for Uniswap v3, we did not observe any new protocol-level considerations, relative to the existing, deployed, and heavily used (over 50K transactions per month) current UAV. Although it is rather awkward for us to be offering protocol-level comments over widely used functionality, we believe that, as part of a comprehensive security audit, we should do so, at the danger of sounding a little "too cautious".

The trust model of the protocol is never clearly spelled out. There is not a clear "ultimate authority". Chainlink pricing is considered more trusted for precise prices, but not authoritative enough to supersede Uniswap pricing outside pre-set bounds. The Owner can set Uniswap pricing as the only authority, per case. This is the main idea of the UAV, therefore it is not open to objection. However, there are nuances that we are not certain are completely thought out. The Uniswap price is denominated in USDC (since the ETH price is computed relative to USDC and other assets are relative to ETH), whereas the Chainlink price is over US dollars. Therefore, USDC is trusted to always be equivalent to US dollars. Furthermore, other assets (USDT, TUSD) are, under current deployment parameters, considered to be always equivalent to US dollars, with no ability to supersede this price. This means that the trust base of the oracle includes, at least: Uniswap, USDC, USDT, TUSD. Financial, coding, or governance threats in the above will likely severely affect Compound (or any other user of the Oracle). It may seem extreme to warn of potential issues in some of the most trusted services on the Ethereum blockchain. However, it is wise to be aware of the attack surface, which is not always

obvious. (E.g., USDC is institutionally well-trusted, but, on-chain, its minting is ultimately governed by a single EOA. USDT is alleged to have systemic risks. TUSD is governed by a 2-of-4 multisig.) Considering the abundance of real-world and on-chain information (Chainlink + Uniswap v3 TWAP) in this oracle, it seems to us strange to place blind trust on all three of these stablecoins.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: 01) User or system funds can be lost when third party systems misbehave. 02) DoS, under specific conditions. 03) Part of the functionality becomes unusable due to programming error.
LOW	Examples: 01) Breaking important system invariants, but without apparent consequences. 02) Buggy functionality for trusted users where a workaround exists. 03) Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed”, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Suboptimal code (`UniswapConfig::getTokenConfig`)	**RESOLVED**

The code of `UniswapConfig::getTokenConfig` has the form:

```
        if (i == 0) {
                underlying = underlying00;
                symbolHash = symbolHash00;
                baseUnit = baseUnit00;
                priceSource = priceSource00;
                fixedPrice = fixedPrice00;
                uniswapMarket = uniswapMarket00;
                reporter = reporter00;
                reporterMultiplier = reporterMultiplier00;
        }
        if (i == 1) {
                underlying = underlying01;
                symbolHash = symbolHash01;
                baseUnit = baseUnit01;
                priceSource = priceSource01;
                fixedPrice = fixedPrice01;
                uniswapMarket = uniswapMarket01;
                reporter = reporter01;
                reporterMultiplier = reporterMultiplier01;
        }
        … // 35 cases
```

Given the large number of cases, turning the `ifs` into `else-ifs` will result in non-negligible savings.

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend addressing them.

ID	Description	STATUS
A1	Unused code	**RESOLVED**

In `UniswapLib.sol`, the `struct Slot0` definition is not being used. It is recommended that it be removed as it is dead code.

ID	Description	STATUS
A2	Code simplification	**RESOLVED**

In `UniswapConfig.sol`, all `getTokenConfigBy*` functions have a check that the index is not `type(uint).max`, however this is redundant as `getTokenConfig` already covers this case by checking that `index < numTokens`.

For example:

```
function getTokenConfigBySymbolHash(bytes32 symbolHash) public view returns
(TokenConfig memory) {
    uint index = getSymbolHashIndex(symbolHash);

    // Dedaub: Redundant check; getTokenConfig checks that index < numTokens.
    //       That check covers the case where index == type(uint).max
    if (index != type(uint).max) {
        return getTokenConfig(index);
    }
    revert("token config not found");
}
```

Can be simplified to:

```
function getTokenConfigBySymbolHash(bytes32 symbolHash) public view returns
(TokenConfig memory) {
      uint index = getSymbolHashIndex(symbolHash);
```

```
        return getTokenConfig(index);
}
```

A3	Redundant trailing modifier parentheses	**DISMISSED**

There are a couple of instances where even zero-argument modifiers are used with parentheses, even though they can be omitted. For example, in `UniswapAnchoredView::activateFailover`:

```
function activateFailover(bytes32 symbolHash) external onlyOwner() {
    ...
}
```

This pattern can be found in:

- UniswapAnchoredView::activateFailover
- UniswapAnchoredView::deactivateFailover
- Ownable::transferOwnership

A4	Reporter sanity check for fixed price assets	**RESOLVED**

In the UniswapAnchoredView constructor, fixed price assets (either ETH or USD pegged) check that the provided uniswap market is zero, however the reporter field is unchecked. It is recommended that the reporter be also required to be zero, for consistency:

```
else {
    require(uniswapMarket == address(0), "only reported prices utilize an
anchor");

    // Dedaub: Check that reporter is also 0
    require(config.reporter == address(0), "only reported prices utilize a
reporter");
}
```

A5	Key functionality is cryptic (`fetchAnchorPrice`)	**RESOLVED**

The correctness of the calculation in `UniswapAnchoredView::fetchAnchorPrice` is very hard to establish. More comments would help. Specifically, the code reads:

```
function fetchAnchorPrice(TokenConfig memory config, uint conversionFactor)
 internal virtual view returns (uint) {
    uint256 twap = getUniswapTwap(config);
    uint rawUniswapPriceMantissa = twap;
    uint unscaledPriceMantissa = rawUniswapPriceMantissa * conversionFactor;
    uint anchorPrice = unscaledPriceMantissa * config.baseUnit / ethBaseUnit /
expScale;
    return anchorPrice;
}
```

The correctness of this calculation depends on the following understanding, which should be documented in code comments, or the functionality is entirely cryptic. (We note that the original UAV code had similar comments, although the ones below are our own.)

- `getUniswapTwap` returns the price between the baseUnits of the two tokens in a pair, scaled to e18
- `rawUniswapPriceMantissa * config.baseUnit` : price of 1 token (instead of one baseUnit of token), relative to baseUnit of the other token. Still scaled at e18
- `unscaledPriceMantissa * config.baseUnit / expScale` : (mathematically, not in integer arithmetic) price of 1 token relative to baseUnit of the other, scaled at 1
- `unscaledPriceMantissa * conversionFactor * config.baseUnit / ethBaseUnit / expScale` :
    - in the case of ETH-USDC, `conversionFactor` is `ethBaseUnit`, and the above happens to return 1 ETH's price in USDC with 6 decimals of precision, just because the USDC unit has 6 decimals
    - in the case of other tokens, the conversionFactor is the 6-decimal ETH-USDC price, hence the result is the price of 1 token relative to 1 ETH, at 6-decimal precision.

A6	Compiler warning	**RESOLVED**

The Solidity compiler is issuing a warning for the `UniswapAnchoredView::priceInternal` function, that the return variable may be unassigned. While this is a false warning, it can be easily suppressed with a simple refactoring of the form:

```
function priceInternal(TokenConfig memory config) internal view returns (uint) {
```

```
    if (config.priceSource == PriceSource.REPORTER) return
prices[config.symbolHash].price
    else if (config.priceSource == PriceSource.FIXED_USD) return
config.fixedPrice;
    else {
        uint usdPerEth = prices[ethHash].price;
        require(usdPerEth > 0, "ETH price not set, cannot convert to dollars");
        return usdPerEth * config.fixedPrice / ethBaseUnit;
    }
}
```

A7	Redundant code (`UniswapConfig::getTokenConfig`)	**RESOLVED**

The expression:

```
((isUniswapReversed >> i) & uint256(1)) == 1 ? true : false
```

can be shortened to the more elegant:

```
((isUniswapReversed >> i) & uint256(1)) == 1
```

A8	Floating pragma	**RESOLVED**

The floating pragma `pragma solidity ^0.8.7;` is used in most contracts, allowing them to be compiled with any version of the Solidity compiler v0.8.* after, and including, v0.8.7. Although the differences between these versions are small, floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contract deployment (Solidity version 0.8.7 at the audit commit hash).

A9	Compiler known issues	**INFO**

The contracts were compiled with the Solidity compiler v0.8.7 which, at the time of writing, have some known bugs. We inspected the bugs listed for version 0.8.7 and concluded that the subject code is unaffected.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.