Unit Protocol Report No. 1

Prepared by Level K

Jan, 2021

Chapter 1

Introduction

1.1 Scope of Work

This code review was prepared by Level K at the request of the Unit Protocol developers. Unit Protocol is developing a "stablecoin" crypto-asset on Ethereum that uses other crypto-assets as collateral for the synthetic "stable" asset. The code covered by this review (see section 1.2) implements the core logic for determining when and how to issue synthetic assets (`USDP`) and under what circumstances the collateral should be sold for `USDP`.

1.2 Source Files

This audit covers code from two public `git` repositories in the `unitprotocol` GitHub organization:

```
core    7ef6ac5740002dd468226db56de41eed4ed8d543
oracles 6895af05dec70445e72402135c9907a87158950f
```

Within `github.com/unitprotocol/core`, only the following files were reviewed:

- contracts/ParametersBatchUpdater.sol
- contracts/USDP.sol
- contracts/Vault.sol
- contracts/VaultParameters.sol
- contracts/liquidators/LiquidationAuction01.sol
- contracts/liquidators/LiquidationTriggerKeep3rMainAsset.sol
- contracts/liquidators/LiquidationTriggerKeep3rPoolToken.sol

- contracts/liquidators/LiquidationTriggerSimple.sol
- contracts/oracles/OracleSimple.sol
- contracts/vault-managers/VaultManagerKeep3rMainAsset.sol
- contracts/vault-managers/VaultManagerKeep3rPoolToken.sol
- vault-managers/VaultManagerParameters.sol
- vault-managers/VaultManagerStandard.sol

Within `github.com/unitprotocol/oracles`, only the following files were reviewed:

- contracts/impl/ChainlinkedKeep3rV10OracleMainAsset.sol
- contracts/impl/ChainlinkedKeep3rV10OraclePoolToken.sol

This review was conducted under the optimistic assumption that all of the supporting software infrastructure necessary for the deployment and operation of the reviewed code works as intended. There may be critical defects in code outside of the scope of this review that could render deployed smart contracts inoperable or exploitable.

1.3 Additional Documentation

Reviewers used the "white paper" architectural description available at `https://github.com/unitprotocol/protocol_docs/raw/master/unit_wp.pdf` as additional documentation to evaluate the source code. The document was accessed most recently on December 28th, 2020, and it had the following `SHA512` hash:

```
aab61c7d12dbbeda50c355721f58846e7228b84bb3f57668fc8b2cd14a4f5b5f
228d78f15e6547ba118127d101a4f1bf1cbdc3eb484ce11e54aba2a3881cf0e9
```

1.4 License and Disclaimer of Warranty

This source code review is not an endorsement of the code or its suitability for any legal/regulatory regime, and it is not intended as a definitive or exhaustive list of defects. This document is provided expressly for the benefit of Unit Protocol developers and only under the following terms:

THIS REVIEW IS PROVIDED BY LEVELK “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LEVELK OR ITS OWNERS, EMPLOYEES, OR SUBCONTRACTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF

SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS REPORT OR REVIEWED SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 2

Architectural Discussion

This chapter provides a brief overview of the Unit Protocol architecture and a discussion of some possible risks associated with the design and configuration of the reviewed code.

2.1 Architectural Overview

Broadly speaking, there are two classes of fiat-pegged stablecoins: those issued by organization who maintain traditional bank deposits that can be redeemed in-kind against stablecoin issuance, and those issued by smart contracts that maintain cryptocurrency deposits that can be redeemed in architecturally-defined circumstances. `USDP` falls into the latter category, along with another popular project called `DAI`. The soundness of fiat-pegged stablecoins depends on their ability to algorithmically ensure that the issued tokens can be redeemed for precisely the correct amount of collateral assets as denominated in the target fiat currency. For `USDP` specifically, that target fiat currency is the US dollar. The Unit Protocol contracts are designed to support arbitrary ERC20-conforming tokens as assets against which `USDP` can be borrowed, but at least `WETH` and `COL` (the Unit Protocol "platform token") are supported. The rest of this chapter will discuss circumstances under which the Unit Protocol architecture may cause the dollar-denominated price of `USDP` to deviate from parity.

2.2 Sensitivity to Governance and Market Conditions

The dollar-denominated price of `USDP` will ultimately be determined by what the market determines the future value of all the deposited collateral in the `Vault` contract will be under liquidation conditions. Since the parameters that determine how much collateral must be present in the `Vault` contract are determined through governance outside of the scope of this review, we cannot provide a complete quantitative analysis of that behavior. However, we can still make some qualitative statements about how certain governance changes would affect the dollar price of `USDP`.

One of the most important economic considerations for any exchange that occurs on the Ethereum blockchain is the cost-of-carry for any position that lasts longer than one transaction. Since the Ethereum block time is currently about 15 seconds, and since most `USDP` holders will likely not mine blocks themselves, it is nearly impossible to guarantee that a particular transaction is executed in a timely fashion. Concretely, a transaction intended to open or close a `USDP` position doesn't execute immediately; it executes at some point about 15 to 30 seconds in the future (or sometimes much longer). More generally, every transaction executed in response to "off-chain" prices has an embedded futures contract of short but indeterminate duration. When the expected price of on-chain assets (as denominated in off-chain assets) is volatile, the value of this implied futures contract becomes relatively more important when considering what ought to be the market-clearing price for a particular on-chain asset. Concretely, the value of `USDP` is not the *current* value of the collateral pool at the price determined by the liquidation contract, but rather the expected value of that price about 30 seconds in the future, minus any adjustments for risk aversion. Moreover, there are circumstances under which transaction fees could become a serious economic consideration for someone trying to exchange `USDP` for other assets (see section 3.3).

Exacerbating the situation above is the fact that the two `Oracle` contracts covered by this review do not provide perfectly up-to-date asset prices. For example, the ETH-to-USD oracle currently employed by the code will use prices that are up to 2.5 hours stale. The largest historical 1-hour swing in ETH-to-USD prices was more than 34%, which means that worst-case drift in on- versus off-chain prices can be economically significant. Users should expect that the price of `USDP` will deviate significantly from $1.00 when there is a large disagreement between the prevailing market price for collateral assets and those same prices as determined by the oracle contracts.

The magnitude of the price deviation between `USDP` and its dollar peg will additionally be influenced by the magnitude of the distance between the total dollar value of the collateral pool and the dollar value of the circulating supply of `USDP`. As the price of a particular position's collateral reserve approaches the price at which it can be liquidated (by anyone), the owner of the position will have an incentive to close out the position by buying `USDP` and repaying it to the `Vault` contract to redeem the collateral before either volatility or systematic price movement lead to the account becoming under-collateralized. Consequently, the price of `USDP` will *increase* as the price of the total collateral pool falls (approaching the average liquidation ratio), and then *decrease* as the collateral falls past the liquidation ratio, since the dollar-denominated value of the assets that can be "bought" from the liquidation auction will fall similarly. Empirical evidence, such as the price spike of `DAI` in March of 2020, increases our confidence in this theoretical analysis.

The market value of `USDP` will be sensitive to a number of parameters decided by the contracts' governance. These include:

- The set of assets that can serve as collateral
- The respective "liquidation ratios" of the collateral assets
- The "oracle" (or set of "oracles") that act as price feeds for assets and the anticipated accuracy of those implementations

- The "devaluation period", or the period over which the price of liquidated collateral is reduced until it is sold
- The "stability fee," or the fee charged against outstanding USDP positions over time

Additionally, the market price of USDP will be sensitive to a number of conditions beyond the direct control of Unit Protocol governance, such as:

- The relative composition of assets comprising the deposited collateral
- The current price level of each collateral asset, denominated in dollars
- The expected price volatility of each collateral asset
- The combined price of collateral assets relative to the circulating supply of USDP, particularly when close to the "liquidation ratio"
- Gas prices and the level of congestion in the Ethereum blockchain, as well as any other factors that would influence transaction timeliness
- Timeliness of oracle price feeds

Given the diversity of conditions that could cause the price of USDP to drift meaningfully from its target, we expect that governance parameters and perhaps even core architectural assumptions will have to be periodically re-visited in order to reduce the tracking error of USDP over time. Our understanding is that the Unit Protocol developers are aware of these risks, as many of them are mentioned explicitly in the design documentation (see section 1.3). The price performance history of DAI indicates that price deviations in excess of 5% from the target would not be unprecedented.

2.3 Systematic Risk

The fact that USDP will be collateralized by a basket of assets rather than a single asset means that the correlation of price movement across assets is a significant factor in determining the risk that collective price movements exceed the rate at which the oracle contracts can be updated to reflect the correct price of the collateral pool. Under a simple model like Modern Portfolio Theory, the standard deviation of returns is minimized as the correlation between assets is reduced. Since the price of USDP is sensitive to the volatility in the dollar-denominated price of the collateral pool, it will be sensitive to the correlation of price movement between collateral pool assets. Moreover, there are certain classes of assets that have prices that move *deterministically* when the price of USDP changes, and those are particularly problematic.

As a pathological example, consider a Uniswap Liquidity Provider token ("LP" token) that is minted when liquidity is provided to a USDP-WETH pair. For notational convenience, we will denote this as LP(USDP-WETH). The price of LP(USDP-WETH) will be sensitive to changes in the price of USDP, since changes in price imply that the

value of the assets held within the pair contract have changed. An asset like `LP(USDP-WETH)` would be catastrophic to allow as part of the `USDP` collateral pool, because changes in the price of `USDP` would **deterministically** reduce the value of the collateral pool, which ought to lead to changes in the price of `USDP`, which would again reduce the value of the collateral pool, and so on. This behavior is magnified when you consider higher-order derivatives like `LP(USDP-LP(USDP-WETH))` or `LP(WETH-LP(USDP-WETH))`.

The governance for `USDP` needs to be diligent about analyzing the assets that are allowed to be part of the collateral pool to determine if they would introduce the sort of pathologically unstable price behavior described here.

Chapter 3

Code Issues

Issues discussed in this section are implementation details that could lead to security vulnerabilities, unintended behavior, or software maintenance problems. The issues in this section are ordered from most- to least-severe based on the judgment of the reviewers.

Users should note that the address that deploys the core Unit Protocol contracts has complete control over economically significant governance features. Malicious governance can drain any and all account balances held in the contracts through the manipulation of contract permissions. This review was conducted under the assumption that governance acts in the best interest of end-users.

3.1 Incorrect Oracle Used in Pool Token Liquidator

In `LiquidationTriggerKeep3rPoolToken.sol`, a price oracle for Uniswap LP tokens is used to price the `vault.col()` portion of the collateral. Note that this particular implementation of `assetToUsd()` only handles Uniswap LP tokens, so we expect this call on line 50 to revert:

```
// USD value of the COL amount of a position
uint colUsdValue_q112 =
    keep3rOraclePoolToken.assetToUsd(vault.col(), vault.colToken(asset,
user));
```

Note that `vault.col()` is presumed to be `COL` or `DUCK`, neither of which are LP tokens. Practically speaking this means liquidation of positions that use this price oracle will always fail.

This bug, as well as another serious bug fixed prior to this audit (in commit `9d5aa139`) were both caused by the fact that each "liquidator" contract is responsible for correctly identifying which oracles are relevant to evaluating the collateral value of a debt position. A new liquidator contract introduced in the future could easily re-introduce either of these bugs.

Since oracles are logically bound to a particular `(asset, user)` tuple, and since the rules for collateral ought to be identical regardless of which "liquidator" contract is used, one could imagine instead the code looking like this:

```
uint asset_q112 = vault.oracleFor(asset, user).assetToUsd(asset, amount);
```

where `Vault.oracleFor(asset address, user address)` is a function that determines which oracle contract is appropriate for determining the value of a particular position. Alternatively, the `Vault` contract could simply consult the appropriate oracle internally, so instead we would have:

```
// liquidation pricing logic:

uint asset_q112 = vault.assetToUsd(user, asset, amount);

// Vault.assetToUsd() implemented as:

function assetToUsd(address user, address asset, uint amount) pub-
lic {
    // oracles[asset][user] set when position is spawned:
    address oracle = oracles[asset][user];
    require(oracle != 0);
    return OracleSimple(oracle).assetToUsd(asset, amount);
}
```

Either of these patterns would make it substantially more difficult for a developer to inadvertently use the wrong oracle (as in this bug) or unintentionally allow the wrong oracle to be used by an attacker (as in commit `9d5aa139`).

3.2 Integer Underflow in Main Oracle

In `ChainlinkedKeep3rV1OracleMainAsset.sol`, the arithmetic on line 100 can underflow, causing unintended behavior:

```
while (block.timestamp - timestampObs < minObservationTimeBack) {
    observationIndex -= 1; // <-- underflow here
    // ...
}
```

There is no guarantee that the loop condition here will be satisfied by the time `observationIndex` is 0, which means `observationIndex` could underflow. The impact of this bug is likely low, as the call on line 101 to `observations()` ought to revert when it is passed an invalid `observationIndex`. However, future changes to this code could make this unintended behavior exploitable.

3.3 Liquidation Front-running

The `LiquidationAuction01.buyout()` function can be called by any address that owns enough `USDP` to satisfy the current price of the assets. (And, since `USDP` will presumably be available to trade on Uniswap, anyone at all will likely be able to borrow `USDP` to satisfy a call to `buyout()` as part of a flash loan.) Since the price paid by the caller to `buyout()` is determined by the number of blocks elapsed since `Vault.liquidate()` was called on the position in question, we expect that many simultaneous calls to `buyout()` will be made once the price of the liquidated assets falls below their market-clearing price. The account that wins this "race" is the only account that benefits economically from calling `buyout()`; the other accounts will forfeit the gas that they have paid for their transactions. Consequently, this race serves as a kind of tax on liquidation that is paid by liquidators and accrues only to miners. It is reasonable to expect that the price of `USDP` could sit below its dollar target if users expect that it will be expensive to successfully redeem liquidated assets.

3.4 Assets Permissions cannot be Revoked

Once governance has allowed an asset to be used as collateral through `VaultParameters`, there is no way for governance to prevent that asset from being used for new debt without breaking `Vault.triggerLiquidation()`. In other words, there is currently no safe way for governance to "turn off" the use of a particular asset as collateral. It would be prudent to add functionality for governance to prevent a particular asset's balance from increasing in the collateral pool. As discussed in section 2.3, certain assets may be inappropriate as collateral.

3.5 Missing Assertion in `COL` Repayment

The `repayUsingCol()` function is the only function in the `widthraw*()` and `repay*()` family of functions that *does not* explicitly close out a debt position and doesn't have a statement like the following:

```
require(debt != 0 && usdpAmount != debt, "Unit Protocol: USE_REPAY_ALL_INSTEAD");
```

Either `repayUsingCol()` should call `Vault.destory()` if the remaining debt is zero at the end of the transaction, or it should include the assertion statement above. Otherwise, the account information in the `Vault` contract may end up in an inconsistent state.

3.6 Superfluous External Calls to Self

The `VaultParameters` contract is instantiated with `Auth(address(this))`, which means that modifiers like `onlyManager()` and `onlyVault()` make external calls dur-

ing evaluation, which increases gas costs. Consider using dedicated modifiers for this contract that access contract-local storage directly.

3.7 Suboptimal Visibility Specifiers

There are many functions in the reviewed code that have the `public` visibility specifier where the cheaper `external` visibility specifier could be used instead. The `public` visibility specifier implies the function needs to deal with two calling conventions, while `external` only implies one, so the generated bytecode for a `public` function is often larger, all else being equal.

3.8 Widespread Code Repetition

Many contracts and functions throughout the codebase repeat pieces of logic that are present in other pieces of code. For example, the files `VaultManagerKeep3rMainAsset.sol` and `VaultManagerKeep3rPoolToken.sol` are each over 100 lines long, but they are substantially identical except for the additional `*_Eth()` family of functions in one contract and the name of the `oracle` identifier. Similarly, the files `LiquidationTriggerKeep3rMainAsset` and `LiquidationTriggerKeep3rPoolToken` are identical except for one integer constant and the naming of some identifiers. Within `VaultManagerKeep3rMainAsset.sol`, the functions `withdrawAndRepayUsingCol()`, `withdrawAndRepayUsingCol_Eth()`, `withdrawAndRepay_Eth()`, and `withdrawAndRepay()` share very similar business logic. For example, the following fee calculation logic on lines 288 through 290 of `VaultManagerKeep3rMainAsset.sol` is repeated almost verbatim on lines 344, 405, 217, and 263, with the only differentiating factor being whether or not the asset in question is `COL` or the main collateral asset type.

```
uint fee = vault.calculateFee(asset, msg.sender, usdpAmount);
uint feeInCol = fee.mul(Q112).div(colUsdPrice_q112);
vault.chargeFee(vault.col(), msg.sender, feeInCol);
```

Repeating security-critical business logic makes it substantially more likely that developers unintentionally introduce bugs as part of regular maintenance. For each piece of repeated business logic, a bug has to be fixed not just once, but once for each place in which that logic is present. Moreover, decreasing the total lines of code of a project is one of the most reliable way of reducing a project's total bug count. The refactoring suggested in section 3.1 ought to eliminate some of the code duplication currently present in the code, because much of the duplication is caused by the fact that a different "vault manager" and "liquidation trigger" is used for each oracle type. If the oracle logic is appropriately abstracted away from liquidation and vault management logic, then that code won't need to be duplicated for each new oracle type. (Even without refactoring the `Vault` as suggested in section 3.1, much of the repeated code could still be refactored into a library of shared logic that is used by many contracts.)

3.9 Granular Vault Permissions

Access to the `Vault` contract is mediated through the `hasVaultAccess()` modifier. However, there are a couple disjoint subsets of `Vault` functionality that are employed by "vault managers" and "liquidators," respectively. Consider granting only the access that is absolutely required for each piece of functionality in order to reduce ease of exploitability of a bug in another contract. For example, only the "liquidator" family of contracts call `Vault.liquidate()`, so the modifier for `Vault.liquidate()` should be `onlyLiquidator()`. Similarly, `borrow()`, `spawn()`, `repay()`, and so forth are only used by "vault manager" contracts, so those functions should be declared as `onlyVaultManager()`.

3.10 Misspelled Method Name

The function `ensureCollateralizationTroughProofs()` is misspelled. It should be `ensureCollateralizationThroughProofs()`.