MixBytes()

UNIT PROTOCOL

CORE SMART CONTRACTS

AUDIT REPORT

OCTOBER 01

2020

FOREWORD
TO REPORT

A small bug can cost you millions. **MixBytes** is a team of experienced blockchain engineers that reviews your codebase and helps you avoid potential heavy losses. More than 10 years of expertise in information security and high-load services and 18 000+ lines of audited code speak for themselves. This document outlines our methodology, scope of work, and results. We would like to thank **Unit Protocol** for their trust and opportunity to audit their smart contracts.

CONTENT
DISCLAIMER

This report is public upon the consent of **Unit Protocol**. **MixBytes** is not to be held responsible for any damage arising from or connected with the report. Smart contract security audit does not guarantee an inclusive analysis disclosing all possible errors and vulnerabilities but covers the majority of issues that represent threat to smart contract operation, have been overlooked or should be fixed.

TABLE OF CONTENTS

01 INTRODUCTION TO THE AUDIT

GENERAL PROVISIONS

Unit Protocol is a decentralized borrowing protocol that allows to borrow USDP Stable Coin using a variety of tokens as collateral.

MixBytes was approached by **Unit Protocol** to provide a security assessment of their core smart contracts.

SCOPE OF THE AUDIT

The scope of the audit is smart contracts at **https://github.com/unitprotocol/core/tree/746ea0c83e309b071d6d204bdd9a8e339713098f/contracts**

Audited commit is 746ea0c83e309b071d6d204bdd9a8e339713098f.

Tokenomics and economical stability of the stablecoin are out of scope of this security audit.

Corrections as well as significant improvements were made after the first audit. The re-audit scope - **https://github.com/unitprotocol/core/tree/518a09081aadda6a383f9845837ed7045101e64f/contracts**, excluding:

* subdirectory `test-helpers`
* `contracts/liquidators/LiquidatorUniswapLP.sol`
* `contracts/oracles/ChainlinkedUniswapOracleLP.sol`
* `contracts/vault-managers/VaultManagerUniswapLP.sol`.

02 | SECURITY ASSESSMENT PRINCIPLES

CLASSIFICATION OF ISSUES

CRITICAL

Bugs leading to Ether or token theft, fund access locking or any other loss of Ether/tokens to be transferred to any party (for example, dividends).

MAJOR

Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.

WARNINGS

Bugs that can break the intended contract logic or expose it to DoS attacks.

COMMENTS

Other issues and recommendations reported to/acknowledged by the team.

SECURITY ASSESSMENT METHODOLOGY

Two auditors independently verified the code.

Stages of the audit were as follows:

1. “Blind” manual check of the code and its model
2. “Guided” manual code review
3. Checking the code compliance to customer requirements
4. Discussion of independent audit results
5. Report preparation

03 DETECTED ISSUES

CRITICAL

Not found.

MAJOR

1. **Vault.sol#L160-L161**
   Vault.sol#L176-L177

Position debt is updated, but field `lastUpdate[token][user]` is not. In the future, this may lead to an increase of the position debt.

Status:

FIXED at **518a090**

2. **Vault.sol#L160-L161**
   Vault.sol#L176-L177
   Vault.sol#L190

Changes of `debts[token][user]` and `tokenDebts[token]` will differ quantitatively, although they should be equal (since `tokenDebts` is the sum). Either after the `tokenDebts` update percentages will be lost, or the `lastUpdate[token][user]` update will be missed. We recommend that you either call `update` first, or consider the difference between `getDebt` and `debts[token][user]` when updating `tokenDebts`.

Status:

FIXED at **518a090**

3. **VaultManagerUniswap.sol#L215**

After this call, interest on interest begins to accrue. Moreover, because of the `update` call below `vault.calculateFee` will always return 0 and payment of commission in COL will become impossible.

Status:

FIXED at **3e192db**

WARNINGS

1. Vault.sol#L219

Fee calculated in this manner does not take into account the so-called compounding interest, i.e. interest on interest. If the absence of compounding interest is the desired behavior, then the code still does not work correctly since the interest “materializes” during the call **update**, after which the interest begins to accrue on it, i.e. the compounding is started “manually”. The fact that the number and intervals of `update` calls (all other things being equal, as well as with constant position parameters) directly affect the financial result, which should not be the case, suggests that the code is incorrect.

Let’s assume that initially the debt is 1000 USDP and the fee is 20%.

1. If you call `update` after a year, the financial result will be 1000 + (1000 * 0.2) = 1200
2. If you call `update` after 180 days, the financial result will be 1000 + (1000 * 0.2) * 180/365 = 1098.63. And when you call another one after 185 days (i.e. a year will pass in total), the financial result will be 1098.63 + (1098.63 * 0.2) * 185/365 = 1210.

We received 10 USDP more debt due to the different sequence of calls.

If compounding is the desired behavior, we recommend changing the calculation formula using integration.

Status:

ACKNOWLEDGED

Client: materialization of interest is the desired behavior, since before the position is updated, interest should be calculated at the previous rate. They materialize, and then the stability fee (the updated one) is charged on the debt body.

2. Vault.sol#L80-L81

These changes do not take effect immediately after the change in `parameters`. It is especially important in case of `stabilityFee`, because depending on the time of the `update` call we will receive a different financial result.

We recommend to make the logic of applying changes independent of the frequency and intervals of the `update` call, for example by creating `parameters` snapshots, followed by the time of each change.

Status:

ACKNOWLEDGED

Client: This behavior is desirable. The user retains the current conditions until he issues additional USDP or withdraws the collateral.

3. Vault.sol#L77
Vault.sol#L164

Probably it is not the best place to control your debt. So, for example, it makes no sense to roll back a transaction if the debt has already exceeded `tokenDebtLimit (token)` due to the interest accrual, since this will hide the problem. In addition, the debt reduction in the event of overshoot should always be encouraged and not blocked. We recommend calculating the corresponding bool indicator in `Vault`, and processing it at a higher level, in the `VaultManager`.

Status:

FIXED at **518a090**

4. Vault.sol#L166
Vault.sol#L178

Since a part of the debt is not created in the form of USDP, but in the form of accrued interest, and the payment of the debt is possible only in USDP, it will be impossible to fully repay all open positions due to the lack of USDP. We recommend that you ensure that this is the desired behavior.

Status:

FIXED at **VaultManagerUniswap.sol#L181-L243**

5. VaultManager.sol#L134

Collateral Proportion is not controlled on an ongoing basis and can be anything at the moment, because in case Collateral Proportion is violated, there is no analogue to the liquidation. We recommend that you ensure that this is the desired behavior.

Status:

ACKNOWLEDGED

6. VaultManager.sol#L205-L209

UniswapOracle is used without alternatives, regardless of `oracleType[token][user]`. We recommend that you ensure that this is the desired behavior.

Status:

ACKNOWLEDGED

Client: There will be alternatives in the future.

7. Vault.sol#L188-L189

The code implies the presence of a certain system that carries out the sale of assets of liquidated positions, but there is none in the code of contracts. So either the solution is not complete, or a part of the solution is outside the scope of this audit.

Status:

ACKNOWLEDGED

Client: Initially the liquidation mechanism will be implemented off-chain.

8. Vault.sol#L192

In some rare cases (paying off a large amount of debt with a large fee) subtracting `.sub (amount)` will lead to underflow and rollback of the transaction. We recommend doing the addition first.

Status:

FIXED at 518a090

9. ChainlinkedUniswapOracle.sol#L20

The proposed oracle is a good decentralized solution, but not the best one. In theory, an attack is possible when the exchange rate of a token with low liquidity on Uniswap is held at inadequately low values for 100 blocks (about half an hour). This will allow to unreasonably liquidate a part of positions and receive a liquidation fee. We recommend using a solution that takes into account all liquidity, from centralized exchanges as well. Fortunately, oracles for many tokens based on Chainlink have now appeared.

Status:

ACKNOWLEDGED

Client: We favor a more decentralized delivery model (Uniswap).

10. ChainlinkedUniswapOracle.sol#L67

From the `UniswapOracle` documentation: “The Uniswap-Oracle library is unaudited. Any application that is responsible for value on mainnet should be fully audited”. We recommend auditing the `@keydonix/uniswap-oracle-contracts` library.

Status:

ACKNOWLEDGED

11. ChainlinkedUniswapOracle.sol#L79

The cost of the specified amount of Ether is additionally multiplied by 1e18. I.e. the result of the function for the input data `1 ether` will be equal not to 385, but to `385000000000000000000`. If scale factor 1e18 is used on purpose, we recommend that you explicitly mark the scale factor of the return value. We would like to give a similar recommendation for a number of other values, for example, ratios represented by percentages without a fractional part are actively used. Now all numerical parameters are passed simply as uint and dimension, as well as the scale factor of the values are indistinguishable, which can lead to errors. Unfortunately, Solidity does not allow you to define your own types derived from uint, representing different values of the domain. The simplest solution can be postfixes or variable prefixes indicating the dimension and/or the scale factor of the variable, for example `price_q112`, `cr_percent`.

Status:

FIXED at 518a090

12. VaultManagerStandard.sol#L49
VaultManagerStandard.sol#L72
VaultManagerStandard.sol#L92
VaultManagerUniswap.sol#L72
VaultManagerUniswap.sol#L111
VaultManagerUniswap.sol#L142
VaultManagerUniswap.sol#L195

An attacker has an opportunity to manipulate the debt positions of third parties, though with no obvious financial benefit to himself. We recommend adding authorization.

Status:

FIXED at 518a090

13. Vault.sol#L216-L245

Interest on the use of debt is ignored. We recommend that you ensure that this is the desired behavior.

Status:

ACKNOWLEDGED

Client: The behavior is as planned.

14. LiquidatorUniswap.sol#L99-L115

The liquidator (`msg.sender`) is not rewarded in any way, and its address is not even recorded. We recommend that you ensure that this is the desired behavior.

Status:

ACKNOWLEDGED

Client: The behavior is as planned.

COMMENTS

1. Parameters.sol#L45-L58

We recommend storing collateral parameters in a `token_address => parameter_structure` mapping. This seems a little more correct from the point of view of code design.

Status:

ACKNOWLEDGED

Client: This solution would require redundant code to create interfaces for structure fields. Technically, the current implementation is not inferior to the proposed one.

2. UniswapOracle.sol#L35-L36

We recommend checking the received pair addresses for inequality to 0.

Status:

ACKNOWLEDGED

Client: It is the old version of oracle, it will be completely rewritten.

3. USDP.sol#L13

Despite the absence of identified security issues, we recommend using multiply audited code as a basis, for example from `openzeppelin-solidity`.

Status:

ACKNOWLEDGED

Client: Contract code WETH9 is also verified.

4. USDP.sol#L109

Since the sufficiency of `allowance[from][msg.sender]` is not checked, the control can reach `assert` inside the `sub` function. In normal operation, this should not happen. `assert` are used to control the self-consistency of the code and their activation indicates an error in the code, or their incorrect use. It is recommended to add a sufficiency check `allowance [from] [msg.sender]` before performing the subtraction operation.

Status:

FIXED at a43c85d

5. Upgradability recommended

Apparently, the project will have to be finalized by adding new functions. We recommend that you think over the upgrade mechanism for an already deployed solution, for example using upgradable contracts. In addition, practice shows that it is extremely important for DeFi projects to be able to close security gaps in contracts after deployment.

6. Vault.sol#L33-L40
Vault.sol#L45-L55

We recommend grouping 7 data structures that store information about a position into one structure with 7 fields, addressed by the token address and the address of the position owner. This seems a little more correct from the point of view of code design.

Status:

ACKNOWLEDGED

Client: This solution would require redundant code to create interfaces for structure fields. Technically, the current implementation is not inferior to the proposed one.

7. Vault.sol#L100

This function threatens to break encapsulation, because it does not perform any position status checks. We recommend that you only allow an empty position to be cleared.

Status:

ACKNOWLEDGED

Client: In the Vault contract functions marked with the hasVaultAccess modifier are low-level system functions. They can only be called from other contracts that are allowed access. These contracts keep track of the correctness of function calls in Vault.

8. Vault.sol#L204

It can be inconvenient to manage a large number of items individually.

Status:

ACKNOWLEDGED

Client: If required, this can be solved with a contract on the higher level, which will take an array as input and call this function in a loop.

9. Vault.sol#L218

It is recommended to put this value into a constant next to the definition of `stabilityFee` in Parameters`.

Status:

FIXED at **a43c85d**

10. Liquidator.sol#L95

We recommend that you create an `enum` that describes the different types of oracles.

11. LiquidatorUniswap.sol#L63
LiquidatorUniswap.sol#L91

The fact that the parameters use data types specific to `UniswapOracle` makes it impossible to abstract the oracle.

Status:

ACKNOWLEDGED

Client: It is assumed that Liquidator contracts will be implemented separately for different types of oracles.

12. LiquidatorUniswap.sol#L111

Usually the collateralization ratio refers to the ratio of collateral to debt. Here the opposite is true. This can lead to some confusion at times.

13. LiquidatorUniswap.sol#L32

There is no need to transmit and store the `COL` address, since it can be obtained as `parameters.COL()`.

14. Vault.sol#L287

Leap years are not taken into consideration.

15. Reentrancy guards recommended

We recommend adding reentrancy guards to the public functions of vault managers and liquidators.

04 | CONCLUSION AND RESULTS

The Unit protocol took current ideas from the stablecoin field as a basis and develops them into several directions. The main distinguishing feature is the flexibility of the created collateral debt positions (CDP). Different tokens can be used as collateral, different oracles can be configured, different CDP parameters can be configured. This provides users with more options, but also carries some risks: security risks, risks to the financial stability of the protocol and stablecoin. Tokenomics and economical stability of the stablecoin are out of scope of this security audit.

Due to the limitations of Ethereum, which are also inherent in many other blockchains, the protocol does not always work in real time, which contrasts with centralized solutions. This is the classic tradeoff of decentralization. Since the system is quite flexible and has many “moving parts”, development and auditing were of significant complexity. Several errors were identified related to debt accounting and interest accrual (on debt). The approach to oracles was also redesigned, a simple Uniswap-based oracle was replaced with a more secure one. This will prevent a hack similar to the first bZx protocol hack. A lot of work has been done and many of the recommended improvements have been made. After a lot of revisions, a full re-audit has been performed.

The following commit does not have any vulnerabilities according to our analysis: **518a090**.

ABOUT MIXBYTES

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build open-source solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, consult universities and enterprises, do research, publish articles and documentation.

Stack

Blockchains

JOIN US