# UMA Audit – Phase 2

## Financial Templates

OPENZEPPELIN SECURITY | MAY 12, 2020

Security Audits

UMA is a platform that allows users to enter trust-minimized financial contracts on the Ethereum blockchain. We previously audited the decentralized oracle component of the system. In this audit we reviewed a particular financial contract template that can be used within the system.

The audited commit is `e6eaa48124ae3f209fb117cf05eb18292cf26d21` and the scope includes all contracts in the `financial-templates` directory. The `WETH9` contract was not included, since it is used only for testing.

Additionally, it should be noted that the financial template design depends on a number of economic and game-theoretic arguments and assumptions. These were explored to the extent that they clarified the intention of the code base, but we did not audit the mechanism design itself.

All external code and contract dependencies were assumed to work as documented.

### Update

All issues listed below have been fixed or accepted by the UMA team. During the review of the fixes for this audit, the UMA team made four additional changes to their code base. These were addressed in PR#1334, PR#1356, PR#1368 and PR#1395.

Our analysis of the mitigations assumes any pending pull request will be merged, but disregards all other unrelated changes to the code base.

Overall, we are satisfied with the security posture of the team and the code base's overall health. As with the oracle audit, we are pleased to see the use of encapsulated functions and well-documented contracts. We must also highlight the team's clear and transparent analysis of known risks, particularly in a system with a novel design.

We would also like to take this opportunity, like the previous audit, to highlight the UMA team's outstanding software development practices in terms of mandatory code reviews, documentation and testing. As before, the reviewed fixes were implemented in encapsulated pull requests that included associated unit tests as well as design discussions occurring in public.

## System overview

Users of the UMA system can deploy new financial contracts that issue synthetic ERC20 tokens that derive value from an external asset. In the audited template, there are whitelists for the supported underlying asset pairs, expiry dates and acceptable ERC20 collateral tokens. The asset whitelist is managed by the `Governor` contract, described in our previous audit, and the collateral whitelist is set during deployment. Any user can choose a combination of these values to deploy a new financial contract.

Subsequently, anyone can take an over-collateralized loan of freshly minted synthetic tokens that will be redeemable for the value of the specified asset at the expiry date, paid in the collateral currency. At expiry, the loan can be repaid with synthetic tokens or by withholding some of the collateral. The synthetic tokens themselves can be freely traded.

One novel, and fundamental, feature of this system is the ability to encourage borrowers to maintain sufficient collateralization without on-chain access to a live price feed. This is achieved by limiting the amount of collateral that can be immediately withdrawn from a position and rewarding liquidators who find and challenge under-collateralized loans. If the liquidation itself is disputed, the price will be retrieved from the UMA oracle to resolve the dispute.

While we encourage innovative designs, the unknown price constraint introduces some surprising risks and behaviors that users should understand before participating. Many of them are listed in UMA's documentation. We would additionally like to highlight:

audited template, these deadlines must be set to one hour.

- Any position, no matter how collateralized, can be liquidated (and immediately closed). If the liquidation is not disputed, all collateral will be lost. This means that even over-collateralized borrowers must constantly monitor their position, or rely on external dispute bots to challenge improper liquidations.
- UMA oracle fees are charged on the deposited collateral, not on the value of borrowed tokens. This introduces a disincentive for over-collateralization. The fees themselves are managed by the UMA governance mechanism, which must be trusted to choose and change them fairly.
- The deposit and withdrawal limits are restricted by the overall global collateralization ratio, which is somewhat manipulable and is expected to exceed the actual collateralization requirement.
- The UMA oracle has an indefinite resolve time, because failed votes can be repeatedly deferred until the next voting round. During this time, all open positions are frozen but they continue accruing fees.

## Ecosystem dependencies

As the ecosystem becomes more interconnected, understanding the external dependencies and assumptions has become an increasingly crucial component of system security. To this end, we would like to discuss how the financial contracts depend on the surrounding ecosystem.

Like the oracle, the financial template does not interact with external projects but it uses time-based logic, which means it is dependent on the availability of Ethereum in multiple ways:

- Undesirable actions, such as withdrawing too much collateral or liquidating colleteralized loans, are discouraged by the threat of these actions being noticed and corrected within an hour. If they are not detected or the transactions are not processed in time, some users will lose their collateral.
- Similarly, financial contracts must pay fees at least every week, on a rolling window, to avoid a late penalty.
- Additionally, as the deadline approaches, it becomes possible for miners to delay payment actions in order to collect this fee.

result in higher loss (in addition to the late penalty, if it applies).

## Critical severity

None.

## High severity

### [H01] Insolvent initial position

The initial sponsor in any contract is free to open a position with any amount of tokens for any amount of collateral, even if it would be insolvent. Since it is non-economical to liquidate an insolvent position, they will not be liquidated. Although, the UMA team intends to liquidate insolvent positions, some discretion would be necessary (or UMA will become a money pump).

This has two important consequences. Firstly, if the sponsor sold their synthetic tokens for the nominal face value, the recipient would be unable to settle them upon contract expiration. Secondly, if there are any other users with open positions when the contract expires, their collateral becomes vulnerable to theft by the initial sponsor.

In both cases, the issue could be prevented by due diligence. Yet if all traders and borrowers were expected to check the collateralization ratio before using synthetic tokens, it would severely limit the usability of the system.

Unfortunately, the described situation can occur multiple times in the same financial contract if all positions are closed and a new one is opened. Nevertheless, the system must prevent insolvent contracts in order to function correctly.

Consider submitting a price request before the first position is opened (as well as in subsequent “initial” positions) to ensure it is collateralized. Should this be prohibitive, consider requiring the initial sponsor to submit a liquidation bond that must expire before they receive their tokens.

Alternatively, consider preventing users from completely withdrawing, redeeming or liquidating the final position. This would not prevent an initial insolvent position but it would strengthen the guarantee that a healthy contract will remain collateralized.

> We believe the UX and complexity tradeoffs of a lockup period outweigh the downside of asking new participants to be aware of the solvency of the contract they're participating in. As for a contract returning to this state, token holders can be assured that as long as they hold a single token, this initial state cannot be reached again. Similarly, a sponsor can be assured that collateral introduced while the contract is solvent cannot be taken unfairly […] We do think this issue should be solved, but we think that it would be better done through a more major re-architecture of the contract for V2.

### [H02] Sponsors can force liquidation of insolvent positions

Liquidators specify a maximum collateral-to-token ratio to ensure they do not accidentally liquidate a position that is collateralized. However, they cannot indicate a minimum collateral-to-token ratio. If the liquidation is front-run in such a way that the target position becomes insolvent (not just under-collateralized), the liquidator will end up burning their tokens with insufficient compensation. We have identified two possible attack vectors:

- A) If the position about to be liquidated is the only open position, the sponsor can redeem their tokens (which reduces the global collateralization ratio to zero) and then create an insolvent position.
- B) If the liquidation is processed late enough, the sponsor might simply complete a slow withdrawal before being liquidated.

Consider including a minimum collateral-to-token ratio to prevent liquidators from accidentally liquidating an insolvent position.

**Update:** *Fixed in PR#1351.*

## Medium severity

### [M01] Sponsors can delay liquidations

Any under-collateralized sponsor can delay liquidation attempts by transferring their position to another address that they control before the liquidation transaction targeted at the old address is processed. This would cause the liquidation to fail.

the global collateralization ratio artificially depressed, potentially below the collateralization requirement. In this scenario, sponsors would be able to create under-collateralized positions or withdraw excessive collateral. In the extreme case, sponsors could delay liquidation until they were insolvent.

Consider delaying position transfers for a short time window, thereby treating them similar to slow withdrawals.

**Update:** *Fixed in PR#1314. Transfers of positions are now delayed for the same duration as slow withdrawal requests.*

## [M02] Passed withdrawal request might not be withdrawable

The `requestWithdrawal` function of the `PricelessPositionManager` contract allows sponsors to submit a withdrawal request for a given amount of collateral. A request is considered passed once a certain period of time elapses, and the function ensures that the time at which a request is expected to pass is equal or lower than the contract's expiration time.

In the corner case where the request's pass time is equal to the contract's expiry time, the request will be accepted, even though it will not be possible to actually execute it. This is due to the fact that passed requests are executed by the sponsor calling the `withdrawPassedRequest` function, which can only be called *before* the contract's expiration time (as it is marked with the `onlyPreExpiration` modifier).

To avoid unexpected behaviors during withdrawal of collateral, consider modifying the `requestWithdrawal` function to only accept requests whose pass time will be strictly *before* the contract's expiration time.

**Update:** *Fixed in PR#1386. A comment from the UMA team worth highlighting:*

> It should be noted, however, that the funds aren't locked and can be withdrawn at final settlement. The contract can also undergo emergency shutdown, which will result in pending/passed withdrawal requests being locked until settlement anyway. We think this change makes sense to make the inequalities consistent, but being able to withdraw after the liveness period is, nonetheless, not guaranteed.

provides mechanisms for financial contracts to initialize, add, remove and query the party members.

However, the `ExpiringMultiParty` contract does not inform the registry when its parties change. In fact, the `ExpiringMultiPartyCreator` sets the initial party member to the address that triggers the deployment, whether or not that address is a party to the financial contract.

Consider updating the `Registry` contract whenever the `ExpiringMultiParty` contract's participants change.

**Update:** *Fixed in PR#1353. The participants are no longer tracked in the `Registry` contract. Note that the party-related functions still exist in the `Registry`, but they do not apply to `ExpiringMultiParty` contracts.*

### [M04] Possible mismatch between price identifier and collateral

The `ExpiringMultiPartyCreator` contract allows the deployer to choose the price identifier and collateral token. If these are chosen to be inconsistent with each other, the price returned by the oracle will not match the expected token price, which could confuse users and lead to unexpected behavior. Consider maintaining an approved mapping between collateral tokens and price identifiers to enforce consistency between these values.

**Update**: *Not an issue. There are use cases of the UMA contracts where the price identifier may not match the collateral token. In the words of the UMA team:*

> While many products will have matching price quote currencies and collateral tokens, we want to allow creating exotic / complex risk exposures if users prefer.

## Low severity

### [L01] Violation of Checks-Effects-Interactions pattern

Solidity recommends the usage of the Checks-Effects-Interactions pattern to avoid potential security vulnerabilities, such as reentrancy. However, the `withdrawPassedRequest` function malicious contract (or an ERC777 that executes arbitrary receive hooks), this pattern could lead to a reentrancy scenario.

Even though the external call is expected to be to a trusted contract on the collateral token whitelist, consider always following the “Checks-Effects-Interactions” pattern.

**Update:** *Fixed in PR#1307. Additionally, PR#1334 introduces reentrancy guards to all public and external functions.*

## [L02] Lack of input validation

In the `PricelessPositionManager` contract:

- The constructor does not validate whether `_expirationTimestamp` is in the past.
- The `deposit` function does not validate whether the `collateralAmount` is non-zero. If it is zero, the function will emit the ERC20 events and the `Deposit` event unnecessarily.
- The `requestWithdrawal` function does not validate whether `collateralAmount` exceeds the sponsor’s collateral. In such a scenario the request would be clearly illegitimate and could be reverted.
- The `withdrawPassedRequest` function does not validate whether calling sponsors have an active withdrawal request. If they do not, the function will emit the `RequestWithdrawalExecuted` event unnecessarily.

In the `FeePayer` contract:

- The `payFees` function does not validate whether the `lastPaymentTime` is the current `time`. In other words, it does not validate if a previous fee-paying transaction already occurred in the current block. If it did, the function will emit the `RegularFeesPaid` event unnecessarily.

In the `ExpiringMultiPartyCreator` contract:

- The `VALID_EXPIRATION_TIMESTAMPS` array only initializes 16 out of the 17 elements. This means that zero will be considered a valid expiration timestamp.

Consider implementing the necessary logic where appropriate to validate all relevant parameters.

The UMA oracle reports prices as `int` value, which may be negative. However, the `PricelessPositionManager` contract does not support negative prices and simply replaces negative values with zero. This mismatch may lead to errors or unexpected behavior.

Consider modifying either the oracle or the financial contract so that they are compatible.

**Update:** *This is the expected behavior. In the words of the UMA team:*

> We've decided not to support negative prices in this version of ExpiringMultiParty because the incentives break down when a token becomes a liability rather than an asset. However, we want to leave the DVM a bit more general to allow negative prices for other use cases outside of positively-valued synthetic token contracts. We set a negative price to 0 in this contract to prevent the contract from locking up any funds in the case that an error in parameterization or at the DVM level causes a negative price to be returned. We expect that any price identifier used by the ExpiringMultiParty will always return nonnegative price values from the DVM.

## [L04] Unsafe addition

The `requestWithdrawal` function of the `PricelessPositionManager` contract does not check for overflows when computing the withdrawal request's pass time. Consider using OpenZeppelin's `SafeMath` library for all integer calculations.

**Update:** *Fixed in PR#1304.*

## [L05] Erroneous docstrings and comments

Several docstrings and comments throughout the code base were found to be erroneous. In particular:

- Line 209 of `FeePayer.sol` repeatedly mentions "collateralToRemove" instead of "collateralToAdd".
- Line 213 of `FeePayer.sol` should say "collateral added" instead of "collateral removed".
- Line 215 of `FeePayer.sol` references a non-existent `totalPositionCollateral` variable.

amount might be reduced to the collateral left in the contract.

- Line 70 of `Liquidatable.sol` identifies `minSponsorTokens` as a parameter for the `Liquidatable` constructor, but it is actually a parameter for the constructor of the `PricelessPositionManager` contract.
- Line 415 of `Liquidatable.sol` says “the dispute failed”, even though there was no dispute.

**Update:** *Fixed in PR#1352.*

### [L06] Missing error messages in require statements

There are several `require` statements without error messages in the audited code base. For instance, only three `require` statements include messages in the `PricelessPositionManager` contract.

Consider including specific and informative error messages in all `require` statements to favor readability and ease debugging.

**Update:** *Fixed in PR#1364.*

## Notes & Additional Information

### [N01] Missing upper bound constraint during fee payment

The `payFees` function of the `FeePayer` contract does not compare the profit from corruption value (stored in the `_pfc` local variable) to the total amount of fees paid before dividing them, as is done in the `_payFinalFees` function.

Should `_pfc` be lower than `totalPaid` (a local variable representing the total amount of fees being paid), the subsequent calculation of the `cumulativeFeeMultiplier` value would underflow, reverting the transaction in the SafeMath `sub` operation.

Following the “fail early and loudly” principle, consider explicitly restricting the upper bound of the `totalPaid` local variable.

*transaction. This ensures funds will not be trapped in the contract, but the cumulative fee multiplier will be set to zero and the contract will no longer work as intended. We strongly suggest thoroughly testing this dangerous scenario, in particular ensuring that users can no longer deposit funds into the defunct contract to prevent accidental loss of funds. Additionally, we suggest explicitly checking for this condition, rather than relying on division-by-zero errors to prevent new deposits.*

## [N02] Inefficient timestamp validation

The `_isValidTimestamp` function of the `ExpiringMultiPartyCreator` contract iterates over a fixed-length array of known expiration timestamps to check whether the provided expiration timestamp is valid. However, this iteration might not be efficient in terms of gas costs. For a more efficient lookup of valid timestamps, consider replacing the `VALID_EXPIRATION_TIMESTAMPS` array with a `uint256` to `bool` mapping where accepted timestamps are set to `true`.

**Update:** *Fixed in PR#1308. It should be noted that timestamps are defined using the `uint32` type, but then implicitly casted and stored as `uint256`.*

## [N03] Repeated function calls to read Profit from Corruption value

The `_payFinalFees` function of the `FeePayer` contract is unnecessarily calling the `pfc` function twice. The first time, in line 134, the function is called and its return value stored in a local variable `_pfc`. However, the function does not reuse the local variable and instead calls `pfc` again in line 140.

To reduce gas costs during execution, consider reusing the `_pfc` local variable instead of calling the `pfc` function again.

**Update:** *Fixed in PR#1339.*

## [N04] Undocumented prior approval of ERC20 tokens

There are functions that execute transfers of ERC20 tokens from the caller's balance using the standard `transferFrom` function. The caller, in these cases, must first approve the contract that is moving the funds. Neither the existing external documentation nor the functions' docstrings explicitly state this requirement. In particular:

- The [illegible], [illegible] and [illegible] functions need the `msg.sender` to first approve the contract to move an amount of synthetic tokens.

So as to clearly document their assumptions, consider expanding the functions' docstrings stating that the caller must first approve enough tokens for the functions to work as expected.

**Update:** *Fixed in PR#1319.*

## [N05] Unnecessary public getter for state variable

An inline comment in the `PricelessPositionManager` contract states that the `rawTotalPositionCollateral` state variable should not be used directly. However, the state variable's visibility is currently `public`, and Solidity will therefore create a public getter for it.

To avoid mismatches between code and documentation, consider restricting access by removing the `public` keyword from the `rawTotalPositionCollateral` state variable.

**Update:** *The UMA team decided not to follow our suggestion to ease testing and inspection after deployment.*

## [N06] Lack of indexed parameters in events

The `CreatedExpiringMultiParty` event defined in the `ExpiringMultiPartyCreator` contract, and the `LiquidationWithdrawn` event defined in the `Liquidatable` contract, do not index any of their parameters. Consider indexing event parameters to avoid hindering the task of off-chain services searching and filtering for specific events.

**Update:** *Fixed in PR#1317.*

## [N07] Function missing view modifier

For added readability, consider declaring as `view` the `_computeFinalFees` function of the `FeePayer` contract, since it does not modify the contract's storage.

**Update:** *Fixed in PR#1315.*

called nor used in any way.

Moreover, the contract itself inherits from the `FeePayer` contract which already implements the `_getStore` function with the same purpose.

To favor simplicity, consider removing the `_getStoreAddress` function from the `PricelessPositionManager` contract.

**Update:** *Fixed in PR#1303.*

## [N09] Code repetition

There are some examples of repeated code blocks performing conceptually atomic operations.

In `FeePayer.sol`:

- Lines 114 to 115 and 140 to 141 are both reducing the `cumulativeFeeMultiplier` by a given amount.

In `PricelessPositionManager.sol`:

- Lines 278 to 279 and 292 to 293 are both resetting the withdrawal request.
- Lines 184 to 185 and 316+319 are ensuring individual and global consistency when increasing collateral balances.
- Lines 210+215, 269 to 270, 356+362 and 534+542 are ensuring individual and global consistency when decreasing collateral balances.

To favor reusability, consider refactoring these repeated operations into private functions.

**Update:** *Fixed in PR#1300.*

## [N10] Inconsistent NatSpec usage

The code base uses the Ethereum Natural Specification (NatSpec) format inconsistently.

For example, in the FeePayer contract, the function `_getCollateral` is using single line comments format, other functions like `_getStore` have no comments at all, while others such consider always following the Ethereum Natural Specification format.

**Update:** *Fixed in PR#1345. The UMA team adopted a style guide where the NatSpec format is reserved exclusively for public and external functions. Single line comments are optionally used for the remaining functions.*

## [N11] Missing docstrings in sensitive functions

Some internal and private functions implementing sensitive functionality are not commented. See for example `_reduceSponsorPosition` and `_deleteSponsorPosition` functions in the `PricelessPositionManager` contract. The lack of documentation might hinder reviewers' understanding of the code's intention, which is fundamental to assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance.

Consider thoroughly documenting all non-trivial functions, even if they are not part of the contracts' public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

**Update:** *Fixed in PR#1343.*

## [N12] Naming issues

The following functions and variables may benefit from better naming:

- The `_getCollateral` and `_convertCollateral` functions in `FeePayer` should be renamed to indicate fee-adjustment operations.
- The `payFees` function of the `FeePayer` contract should be renamed to `payRegularFees`, `payOngoingFees`, `payRegularAndPenaltyFees` or similar.
- The `priceIdentifer` variable in `PricelessPositionManager` should be `priceIdentifier`.
- The `EndedSponsor` event in `PricelessPositionManager` should be `EndedSponsorship` or `EndedSponsorPosition`.
- The `create` function of `PricelessPositionManager` should be renamed to indicate the possibility of augmenting an existing position.

accordingly.

**Update:** *Fixed in PR#1341. The `create` function has not been renamed but its docstrings have been improved to better reflect the function's behavior.*

## [N13] TODOs in code

There are "TODO" comments in the code base that should be removed and instead tracked in the project's backlog of issues. See for examples:

- Lines 253 and 478 of `PricelessPositionManager.sol`.
- Line 206 of `Liquidatable.sol`.

**Update:** *Fixed in PR#1327.*

## [N14] Typographical errors

- In `ExpiringMultiPartyCreator.sol`:
- Line 148: "constrainments" should be "constraints".
- In `FeePayer.sol`:
- Line 30: there is an extra space before "Finder".
- Line 82: "more" is misspelled.
- Line 195: "cumulativeFeeMultiplier" is misspelled.
- Line 210: "cumulativeFeeMultiplier" is misspelled.
- In `Liquidatable.sol`:
- Line 95: "multiplier" is misspelled.
- Line 98: "multiplier" is misspelled.
- Line 126: "DisputeSucceeded" should be "disputeSucceeded".
- Line 147: there is an unmatched backtick symbol.
- Line 185: "tokens" is misspelled.
- Line 288: "whose" is misspelled.
- Line 370: the word "is" is missing.
- Line 371: "from times" should be "times from".
- In `PricelessPositionManager.sol`:
- Line 39: "is" should be "as".

- Line 288: “the” should be removed.
- Line 322: “mint the caller” should be “mint to the caller”.
- Line 377: “Burns” should be “burns”.
- In `SyntheticToken.sol`:
- Line 8: there is an extra space before “who”.
- In `TokenFactory.sol`:
- Line 12: it should say “return it to the caller”.
- Line 13: “adding new roles” should be “assigning the roles”.
- Line 16: “tokens” should be “token’s”.

**Update:** *Fixed in PR#1298.*

### [N15] Unnecessary imports

In the `PricelessPositionManager` and `Liquidatable` contracts, consider removing the imports statement for the `Testable` contract, as this contract is never used in any of them. Similarly, consider removing the unused import for the `FixedPoint` library in the `ExpiringMultiParty` contract.

**Update:** *Fixed in PR#1299 and PR#1236.*

## Conclusion

Originally, no critical and 2 high severity issues were found. Some changes were proposed to follow best practices and reduce potential attack surface. We later reviewed all fixes applied by the UMA team and **all the most relevant issues have been already fixed**.

## Related Posts

### Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

### OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

### Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits

**Defender Platform**

Secure Code & Audit
Secure Deploy
Threat Monitoring
Incident Response
Operation and Automation

**Services**

Smart Contract Security Audit
Incident Response
Zero Knowledge Proof Practice

**Learn**

Docs
Ethernaut CTF
Blog

**Company**

About us
Jobs
Blog

**Contracts Library**

**Docs**

Privacy | Terms of Use