UMA Audit – Phase 4

FEBRUARY 2, 2021 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

UMA is a platform that allows users to enter trust-minimized financial contracts on the Ethereum blockchain. We previously audited the decentralized oracle, a particular financial contract template and some ad hoc pull requests. In this audit we reviewed some upgrades to the system, including a new financial contract template and the optimistic oracle. The review was completed by 2 auditors over 2.5 weeks.

The audited commit is `1631ef7ad29aaeba756ef3b9a01c667e1343df85` and the scope includes the following contracts:

- financial-templates/common/FundingRateApplier.sol
- financial-templates/common/financial-product-libraries/FinancialProductLibrary.sol
- financial-templates/common/financial-product-libraries/PreExpirationIdentifierTransformationFinancialProductLibrary.sol
- financial-templates/common/financial-product-libraries/StructureNoteFinancialProductLibrary.sol
- financial-templates/perpetual-multiparty/ConfigStore.sol
- financial-templates/perpetual-multiparty/ConfigStoreInterface.sol
- financial-templates/perpetual-multiparty/Perpetual.sol
- financial-templates/perpetual-multiparty/PerpetualCreator.sol
- financial-templates/perpetual-multiparty/PerpetualLib.sol
- financial-templates/perpetual-multiparty/PerpetualLiquidatable.sol
- financial-templates/perpetual-multiparty/PerpetualPositionManager.sol
- oracle/implementation/OptimisticOracle.sol

We also reviewed the following Pull Requests, as they apply to the previously audited Expiring Multiparty financial template:

- Pull Request 2242
- Pull Request 2176
- Pull Request 2191
- Pull Request 2221
- Pull Request 2119
- Pull Request 2145
- Pull Request 2229
- Pull Request 2239
- Pull Request 2203

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

Summary

As with the previous audits, we are happy with the security posture of the team, the level of documentation and testing, and the code base's overall health. We also appreciate that the project's design discussions are publicly available online.

System overview

The system architecture is still correctly described by our previous audit reports. The main addition is the new Perpetual Multiparty template. This is very similar in purpose and function to the existing Expiring Multiparty template, except there is no expiration date. Instead, synthetic tokens can be issued, traded and redeemed indefinitely (unless the contract goes into emergency shutdown). This improves the generality and usefulness of the template but it introduces a complication. Without a final settlement procedure, the Perpetual Multiparty contracts need a different mechanism to ensure the price tracks the underlying asset.

Conceptually, this is accomplished by continuous funding payments between sponsors and token holders to correct any price discrepancies. In practice, this is achieved by applying an incremental correction to the internal exchange rate between synthetic tokens and the underlying collateral for the purposes of liquidation, or settlement in the event of an emergency shutdown. The particular funding rate is based on an external price monitoring mechanism. Since the UMA Data Verification Mechanism (DVM) resolves too slowly to be used as a live price feed, the funding rate is introduced into the system through a new “optimistic” oracle.

The optimistic oracle is a simple incentive scheme built on the recognition that as long as a price request is well specified and the UMA DVM is functioning correctly, the final resolution is predictable. Instead of waiting for the DVM to resolve, requesters can offer a reward for anyone to post the price immediately, along with a bond. Anyone else can dispute the price and match the bond. If no dispute occurs within a short time window, the optimistic oracle returns the proposed price. If there is a dispute, the oracle reverts to using the DVM, and both bonds are sent to the vindicated party.

Although disputed prices require the full DVM mechanism to resolve, this won't stall any updates to the Perpetual Multiparty template funding mechanism. This is because the contract does not require the external price at any particular timestamp, so it can simply ignore disputed requests until the oracle optimistically resolves with the price at a new timestamp. The reward, which is added to the system usage fees, is only paid to optimistically resolved requests.

Privileged Roles

There are some privileged roles that affect the new features.

Each Perpetual Multiparty contract has configuration parameters that affect how it interacts with the optimistic oracle to update the funding rate. In particular, the contract deployer can choose (and change) the size of the bond required by price proposers and the size of the reward they receive if successful. They also choose the time delay before any changes to these parameters takes effect. Users must trust the contract deployer to choose these parameters wisely, or at least ensure the time delay is long enough that they can exit their positions if they disagree with a pending update.

In this audit we also reviewed a couple of libraries that intermediate requests between the Expiring Multiparty contracts and the oracle. These can be used to dynamically manipulate price requests or the prices themselves at key moments during the life of the contract. In both cases, the library owner can choose the manipulation that applies to the new financial contract, at which point they lose the ability to change it. Users should validate that the configuration is correct and complete before interacting with the system.

Ecosystem dependencies

Like the rest of the system, the reviewed code uses time-based logic, which means it depends on Ethereum availability in multiple ways:

- The Perpetual Multiparty template shares all time-based requirements with the previously described Expiring Multiparty template
- Additionally, optimistic price requests are automatically resolved after the dispute window, so dispute transactions that are sufficiently delayed will not be effective
- Updates to the funding rate, when it applies and how much it compounds are all effected by the timing of state-changing transactions. Any delays will have a minor effect on the resulting calculation

Update

All of the following issues have been addressed or acknowledged by the UMA Team. Our analysis of the mitigations is limited to the specified pull requests and disregards all other unrelated changes to the code base.

While reviewing the updates, the UMA team asked us to include the following unrelated Pull Requests.

- Pull Request 2256
- Pull Request 2262
- Pull Request 2267
- Pull Request 2260
- Pull Request 2278
- Pull Request 2284
- Pull Request 2280
- Pull Request 2287
- Pull Request 2293
- Pull Request 2339
- Pull Request 2395

Since we have minimal feedback, it's included in this section:

- In Pull Request 2262, the comment in the `proposeNewRate` function still explicitly lists the original hardcoded times. This comment is updated in Pull Request 2260 but the 30 minute window is still mentioned.
- In Pull Request 2280, the comment in the `_getLatestFundingRate` function says the code will disregard proposals if the requester address is empty, but it's checking the proposer address.

Critical severity

None.

High severity

[H01] Bond penalty may not apply

The optimistic oracle is designed so that in the event of a dispute, the incorrect party pays the bond penalty to the vindicated party, as determined by the DVM. However, if the proposer and disputer are the same entity, this transfer has no effect. The only remaining deterrent is the DVM fee. Since the bond size is specifically chosen to dissuade attackers from submitting the wrong price and delaying resolution, the ability to nullify the bond penalty undermines the economic reasoning. Moreover, if the reward exceeds the DVM fee, the attacker may actually be positively rewarded for delaying the resolution.

This attack does not apply to contracts deployed from the Perpetual Multiparty template, because they disregard disputed price requests. Nevertheless, it does limit the simplicity and applicability of the optimistic oracle. Consider burning some or all of the bond penalty and ensuring the reward is not high enough to compensate.

Update: *Fixed in PR#2329 and PR#2429. If the `finalFee` associated with the request is non-zero, half the bond is added to the fee and sent to the UMA `Store`. Naturally, this reduces the amount paid to the vindicated party.*

Medium severity

[M01] Lack of event emission after sensitive actions

The `_getLatestFundingRate` function of the `FundingRateApplier` contract does not emit relevant events after executing the sensitive actions of setting the `fundingRate`, `updateTime` and `proposalTime`, and transferring the rewards.

Consider emitting events after sensitive changes take place, to facilitate tracking and notify off-chain clients following the contract's activity.

Update: *Fixed in PR#2284 and PR#2311.*

[M02] Functions with unexpected side-effects

Some functions have side-effects. For example, the `_getLatestFundingRate` function of the `FundingRateApplier` contract might also update the funding rate and send rewards. The `getPrice` function of the `OptimisticOracle` contract might also settle a price request. These side-effect actions are not clear in the name of the functions and are thus unexpected, which could lead to mistakes when the code is modified by new developers not experienced in all the implementation details of the project.

Consider splitting these functions in separate getters and setters. Alternatively, consider renaming the functions to describe all the actions that they perform.

Update: *Fixed in PR#2318.*

Low severity

[L01] Long and complex functions

Some functions are too long and complex. For example, the `createLiquidations` and `withdrawLiquidations` functions of the `PerpetualLiquidatable` contract span around 100 lines. The `_getLatestFundingRate` of the `FundingRateApplier` contract has four levels of nested conditional blocks. This affects readability and maintainability of the code.

Consider refactoring long and complex functions. Blocks of code that require comments can be extracted as internal helper functions with descriptive names. The same can be done with blocks of code inside conditionals. Consider following the top-to-bottom definition of functions and the stepdown rule from the Clean Code book to define every public API function followed by their lower-level helpers, which allows reviewers to easily dig into the implementation details that they care about. Consider measuring the cyclomatic complexity of the code and keeping it low.

Update: *Acknowledged and not changed. UMA's statement for this issue:*

> We have decided not to address this issue because our code style avoids refactoring long methods into internal functions if the internal functions are not reused by other methods. This is primarily to make it easier for a reader to trace calls in the contract — it can be hard to follow calls that jump through many internal methods. For example, we could refactor the code in createLiquidations from line 241-268 into a "_calculateLiquidationParams" internal method, and from 280-298 into a "_pullLiquidationCollateral" internal method, but we think these internal functions could hinder readability for those trying to trace the logic.

[L02] Complex Repay Function

The `repay` function of the `PerpetualPositionManager` contract simply redeems the tokens and then deposits the retrieved collateral. This should have no net effect on the collateral token. This produces clean code but introduces unnecessary fragilities and complexities:

- the `deposit` function pulls the collateral, so the user needs to have previously granted the relevant collateral token allowance to the contract before either having the collateral or knowing the amount of the deposit. Confusingly, they need to do this for an operation that ostensibly has nothing to do with the collateral token.
- the matching transfers correctly result in the same collateral balances, but they unexpectedly emit opposing `Transfer` events
- the transfer may be unexpectedly effected by idiosyncrasies of the collateral token, which could include transfer fees or recipient blacklists (like USDC)
- the `repay` operation unexpectedly emits `Redeem` and `Deposit` events

Consider directly implementing the `repay` function to avoid unnecessarily involving the collateral token and the sponsor's collateral balance.

Update: *Fixed in PR#2309, which also adds the `repay` function to the `PricelessPositionManager`. To reduce bytecode, many error message strings were removed.*

[L03] Current config not updated

In the `getCurrentConfig` of the `ConfigStore` contract, when the pending proposal has passed the `pendingConfig` is returned but the `currentConfig` is not updated. Since `currentConfig` is a private variable this has minimal implications, but it causes a confusing internal state for the contract, which would emit the `ChangedConfigSettings` later than the moment where it starts to be used.

Consider using the `updateConfig` modifier to the `getCurrentConfig` function.

Update: *Fixed in PR#2313.*

[L04] Emergency shutdown defined at the wrong level

The emergency shutdown functionality is defined in the `PerpetualPositionManager` contract, but it is also used in its parent, the `FundingRateApplier` contract. Instead of defining the common functionality at a common base level of inheritance, it is defined at the bottom level and accessed in an unclear way by the parent. Since this parent contract has no access to the emergency shutdown functions and modifiers, it duplicates the code.

Consider extracting the emergency shutdown functionality to a separate contract that gets inherited by the first parent contract that requires it.

Update: *Fixed in PR#2310*.

[L05] Event missing `emit` keyword

In the `createPerpetual` function of the `PerpetualCreator` contract, the `CreatedConfigStore` event is emitted without the `emit` keyword.

Since Solidity v0.4.21, the `emit` keyword has been recommended to distinguish events from other function calls.

Consider using the `emit` keyword for all events.

Update: *Fixed in PR#2303*.

[L06] Inconsistent hasPrice determination

The `hasPrice` function of the `OptimisticOracle` contract returns `true` if the request is either `Settled` or `Resolved`. Although the internal state of the contract does not record the price in the `Resolved` state, this makes sense because retrieving the price will settle the request. This means that from a user's point of view, a `Resolved` price request can be considered to have a price. However, the same reasoning suggests that an `Expired` price request can also be considered to have a price.

Consider adding another condition to the `hasPrice` function so it returns `true` for `Expired` price requests.

Update: *Fixed in PR#2319*.

[L07] Incorrect error message

The `Liquidatable` constructor and the `PerpetualLiquidatable` constructor both revert if the collateral requirement is not greater than 100%, but the error message states “CR is more than 100%”. Consider updating these messages to describe the error condition instead of the success condition.

Update: *Fixed in PR#2304*.

[L08] Misleading comments

The following comments are not accurate:

- In the constructor of the `ConfigStore` contract, the comment says “New settings go into effect after a liveness period passes.” This could be confusing because the constructor directly sets the initial configuration, and the liveness period only applies to the following configuration changes.
- In the `_updateConfig` function of the same contract, the comment says “If liveness has passed, publish new reward rate.” but the `proposerBondPct` and `timelockLiveness` might also have changed.
- In the `setBond` functions of the `OptimisticOracleInterface` and the `OptimisticOracle` contract, the comments say “Requests a new price.”, which is not what these functions do.

- In the `applyFundingRate` function of the `FundingRateApplier` contract, the comment says "This method takes 4 distinct actions", but then it only describes 3 actions.
- In the `transformPriceIdentifier` function of the `FinancialProductLibrary` contract, the comment refers to a `transformedCollateralRequirement` return value, but what is returned is the `priceIdentifier`.
- In the `fromUnscaledUint` function of the `FixedPoint` library, the comment says " `b=5` gets stored internally as `5**18` ", but it is stored as `5*(10**18)`.
- In the `transformPriceIdentifier` function of the `PreExpirationIdentifierTransformationFinancialProductLibrary` contract, the comment says "Else, return the transformed price identifier.", but what is returned is the untransformed price identifier.
- In the `repay` function of the `PerpetualPositionManager` contract, the comment says "numTokens is the number of tokens to be burnt for a commensurate amount of collateral" but the collateral is not refunded.

These comments could cause mistakes for developers relying on them instead of the implementation.

Consider updating the misleading comments.

Update: *Fixed in PR#2308.*

[L09] Missing NatSpec comments

Some of the functions have missing or incomplete Ethereum Natural Specification Format (NatSpec) comments.

- the `publishPendingConfig` function of the `ConfigStore` has no NatSpec comments.
- the `OracleAncillaryInterface` has included an `ancillaryData` parameter to all the function calls, but the NatSpec comments have not been updated accordingly.

Update: *Fixed in PR#2305.*

[L10] Functions not failing early

The `createPerpetual` function of the `PerpetualCreator` contract will run most of its logic before eventually reverting when the `params.collateralAddress` is not approved. Similarly, the `withdraw` function of the `PerpetualPositionManager` contract reverts when `collateralAmount` is equal to zero, but only after already getting the `positionData` using the nontrivial `_getPositionData`.

Consider validating the parameters at the start of the function. This makes the requirements clearer, the helper functions can assume they are getting valid values, and reduces the amount of wasted gas in case of a revert.

Update: *Partially fixed in PR#2306. UMA's statement for the issue:*

> This PR moves the required checks to the top of the function to follow the Checks-Effects-Interactions pattern.
>
> Note that the audit recommends checking that an identifier is registered within the `IdentifierWhitelist` to preserve gas within the `PerpetualCreator` in the case of a reverting call. This change was chosen to not be implemented as this function is not called very often (only by perpetual deployers) so reverting calls will be far less common than other public methods. Additionally, to enable the `PerpetualCreator` to be able to verify if an identifier is registered the `PerpetualCreator` would need to import two additional interfaces (`OracleInterfaces` & `IdentifierWhitelistInterface`) as well as implement additional methods and logic to repeat an existing check that occurs within the constructor of the `PerpetualPositionManager`. This additional complexity & code is not warranted to save a small amount of gas for a perpetual deployer.

[L11] Propose or Dispute a price from the zero address

The Optimistic oracle allows anyone to propose a price or dispute a proposal on behalf of the zero address. Since the state of the incentive mechanism is partially determined by whether the proposer or disputer addresses are non-zero, this won't update the state machine, but it will set variables and emit events prematurely. For example, a client may incorrectly react to a `DisputePrice` event even though the dispute is not recognized by the oracle. Consider restricting the `proposePriceFor` and `disputePriceFor` function parameters to non-zero participants.

Update: *Fixed in PR#2314*.

[L12] Mixed testing and production code

Many contracts, like `FeePayer`, inherit from the `Testable` contract in order to manipulate time during tests. Mixing testing and production code is very risky because mistakes in the testing code could put the production code at risk. It also pollutes the production code making it harder to read.

Consider renaming the `Testable` contract to something like `TimeProvider`. Consider further isolating the testing code by using two different versions of this contract, instead of a single contract that can switch between different modes. The test version should be stored in the `test` directory and the build environment should select between the two contracts.

Update: *Not changed. UMA's statement for the issue:*

> We have decided not to address this issue because we view this suggestion as one that would shift complexity from the contracts to the build/deployment infrastructure. We completely agree that having test infrastructure in production contracts is risky, but we view the simplicity of the Testable contracts as a feature. The Testable contract has been previously audited and battle-tested, so we see modifying it as riskier than leaving it in place.

Notes & Additional Information

[N01] Approximate compounding

In the Perpetual Multiparty template, the funding rate is applied on every fee-accruing transaction, where the total change is simply the rate applied evenly since the last update. This implies that the compounding rate depends on the frequency of update transactions, and it never reaches the ideal of continuous compounding.

This approximate-compounding pattern is common in the Ethereum ecosystem, and the discrepancy is likely unimportant in this case. Nevertheless, in the interest of clarity, we believe it is worth noting. Consider documenting this behavior in the `FundingRateApplier` comments.

Update: *Fixed in PR#2328. To reduce bytecode, some error message strings were removed.*

[N02] Consider ERC777 tokens

The ERC777 token standard extends ERC20 tokens to include send and receive hooks on every balance update. Although the standard is intended to be backwards compatible, it can introduce unexpected risks into systems designed to work with ERC20s.

In this case, the `Liquidatable` contract has been refactored to distribute funds to all participants in the same transaction. The same pattern is used in the `PerpetualLiquidatable` contract. If any of those addresses have a receive hook that reverts, none of the participants could withdraw their funds.

Similarly, if an address that requests a price from the Optimistic oracle prevents the dispute refund transfer, they will disable the dispute functionality.

If the system should support ERC777 tokens, consider isolating the transfer logic so that maliciously constructed send or receive hooks cannot disable any functionality. Otherwise, consider explicitly documenting that ERC777 tokens should not be used as collateral or oracle rewards.

Update: *Fixed in PR#2345 and PR#2347. ERC777 tokens are not currently supported and this has been documented in the contract comments.*

[N03] Incorrect filename

The `StructuredNoteFinancialLibrary` contract is defined in a file named `StructureNoteFinancialProductLibrary`. Consider renaming the file to match the library.

Update: *Fixed in PR#2317.*

[N04] Naming issues hinder code understanding and readability

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are:

- `PreDispute` to `NotDisputed`.
- `PendingDispute` to `Disputed`.
- `disputeBondPct` to `disputeBondPercentage`.
- `sponsorDisputeRewardPct` to `sponsorDisputeRewardPercentage`.
- `disputerDisputeRewardPct` to `disputerDisputeRewardPercentage`.
- `dispute` to `disputeLiquidation`.
- `regularFees` to `paysRegularFees`.
- `_collateralAddress` to `_collateralTokenAddress`.
- `proposeNewRate` to `proposeFundingRate`.
- `tokenCurrency` to `syntheticToken`.
- `Deposit` to `Deposited`.
- `Withdrawal` to `Withdrawn`.
- `RequestWithdrawal` to `WithdrawalRequested`.
- `RequestWithdrawalExecuted` to `WithdrawalExecuted`.
- `RequestWithdrawalCanceled` to `WithdrawalCanceled`.
- `NewSponsor` to `SponsorAdded`.
- `Redeem` to `Redeemed`.
- `EmergencyShutdown` to `EmergencyShutdownEnabled`.
- `SettleEmergencyShutdown` to `EmergencyShutdownSettled`.
- `_getIdentifierWhitelist` to `_getIdentifierApprovedList`.
- `rewardRate` and `proposerBond` in both events to `rewardRatePerSecond` and `proposerBondPercentage` respectively.

Update: *Partially fixed in PR#2332. Per UMA's discussion in the PR thread, there are many instances where a renaming would break infrastructure.*

[N05] Not using the immutable keyword

In line 95 of `PerpetualLiquidatable.sol` a group of immutable state variables are defined. Since Solidity version 0.6.5 the `immutable` keyword was introduced for variables that can only be set once, but these definitions are not using this feature of the compiler.

Consider using the `immutable` keyword for immutable state variables. Note that this change would require updating the contracts to require version 0.6.5 of the compiler or later.

Update: *Acknowledged. UMA's statement for the issue:*

> After implementing this change, we discovered that each use of the immutable keyword adds ~60 bytes per instance to the contract size. This spread over the `PerpetualPositionManageri` & the `PerpetualLiquidatable` means that it would incur a considerable bytecode cost.
>
> As a result, we've chosen to do nothing for this issue for two reasons:
> 1. The extra bytecode.
> 1. Consistency with existing code outside of the scope of the audit.
>
> Note: this change may be considered when the entire repository is upgraded to Solidity 0.8+.

[N06] Redundant code and unnecessary complex calculations

Some functions are duplicating code or performing unnecessary calculations:

- The `hasPrice` function of the `OptimisticOracle` contract can call `getState` twice to determine the return value. Consider saving the output of `getState` in a variable that can be used to determine the return value.

- In the `_applyEffectiveFundingRate` function of the `FundingRateApplier` contract, the `_getLatestFundingRate()` function is unnecessarily called when the value was already assigned to `_latestFundingRatePerSecond`. Given the `_getLatestFundingRate` function has nontrivial logic, consider reusing `_latestFundingRatePerSecond`.

Update: *Fixed in PRs #2319 and #2284.*

[N07] TODOs in code

There are "TODO" comments in the code base that should be tracked in the project's issues backlog. See for example:

- Line 17 of `FundingRateApplier.sol`.
- Line 427 of `PerpetualLiquidatable.sol`.
- Line 532 of `PerpetualPositionManager.sol`.

During development, having well described "TODO" comments will make the process of tracking and solving them easier. Without that information, these comments might tend to rot and important information for the security of the system might be forgotten by the time it is released to production.

These TODO comments should at least have a brief description of the task pending to do, and a link to the corresponding issue in the project repository.

Consider updating the TODO comments, adding a link to the corresponding issue in the project repository. In addition to this, for completeness, a signature and a timestamp can be added. For example:

// TODO: point this at an interface instead.
// https://github.com/UMAprotocol/protocol/issues/XXXX
// –mrice32 – 20201129

Update: *Fixed in PR#2327*

[N08] Typographical errors

- In `FundingRateApplier.sol`:
- line 142: “to” should be removed
- In `PreExpirationIdentifierTransformationFinancialProductLibrary.sol`:
- line 10: “than” should be “then”
- line 21: “Cant” should be “can’t”
- line 23: “exposes” should be “expose”
- In `StructureNoteFinancialProductLibrary.sol`:
- line 14: “determine” should be “determined”
- line 23: “can not” should be “cannot”
- line 25: “exposes” should be “expose”
- In `ConfigStore.sol`:
- line 14: “and” should be “an”
- line 144: “can” should be removed
- In `PerpetualCreator.sol`:
- line 20: there is an extra space
- In `PerpetualLiquidatable.sol`:
- line 545: “valid(liquidation” should be “valid (liquidation”
- In `PerpetualPositionManager.sol`:
- line 229: “witdrawl.” should be “withdrawal.”
- line 242: “withdraw`” should be “withdraw”
- In `OptimisticOracle.sol`:
- line 211: “others'” should be “other’s”
- line 327: “value” should be “valid”
- There are several places where “identifiy” should be “identify”
- In `OptimisticOracleInterface.sol`:
- line 77: “others'” should be “other’s”
- line 152: “value” should be “valid”
- There are several places where “identifiy” should be “identify”

Update: *Partially fixed in PR#2312. All typos were fixed except the one on line 25 of `StructureNoteFinancialProductLibrary.sol`.*

[N09] Unnecessary imports

In the `PerpetualPositionManager.sol` file, consider removing the import statement for `FeePayer.sol`, as it is never used in the `PerpetualPositionManager` contract.

Also, in the `FundingRateApplier.sol` file, consider removing the import statements for `StoreInterface.sol` and `FinderInterface.sol`, as they are never used in the `FundingRateApplier` contract.

Update: *Fixed in PR#2315*.

[N10] Unnecessary type cast

In lines 102 and 104 of `PerpetualCreator.sol` the `derivative` variable is cast to the `address` type. Since it is defined as `address` type, the casts are unnecessary.

Consider omitting this `address` type cast.

Update: *Fixed in PR#2316*.

Conclusion

Originally, no critical and one high severity issue was 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 issues have been fixed or acknowledged.

Security Audits

- If you are interested in smart contract security, you can continue the discussion in our forum, or even better, join the team 🚀
- If you are building a project of your own and would like to request a security audit, please do so here.

RELATED POSTS