OpenZeppelin

UMA Audit Phase 1

Oracle and common

OPENZEPPELIN SECURITY | APRIL 28, 2020

Security Audits

UMA is a platform that allows users to enter trust-minimized financial contracts on the Ethereum blockchain. One component of the system is a decentralized oracle, which was the subject of this audit.

The audited commit is `9d403ddb5f2f07194daefe7da51e0e0a6306f2c4` and the scope includes all production contracts in the following directories:

- the common directory
- the oracle directory

The following directories were not included:

- common/test
- oracle/test
- oracle/implementation/test

Additionally, it should be noted that the oracle 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. Lastly, the oracle

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 highlighted three additional non-severe issues they had identified in their code base. These were addressed in PR#1246, PR#1267 and PR#1262.

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

Here we present our findings, along with the updates for each one of them.

Summary

Overall, we are happy with the security posture of the team and the health of the code base. We are pleased to see the use of mostly small, encapsulated functions and well-documented contracts. We also appreciate that the UMA team has considered the various privileged roles and their security implications.

We must highlight the UMA team's outstanding software development practices in terms of mandatory code reviews, documentation and testing. Their developers hold insightful discussions in the open, which greatly improves the project's transparency and allows their community to witness and get involved in the development process of the protocol. Most, if not all, reviewed fixes were done in encapsulated pull requests, always accompanied by relevant unit tests to confirm behavior where appropriate.

Finally, we applaud the team's effort to have a public bug bounty program.

System overview

Users of the UMA system can enter into financial contracts that distribute funds according to the future price of an asset. Whenever the contract requires the asset price at a given timestamp (for instance, to ensure both parties are still correctly collateralized or to close a disputed contract at expiration), it will send a price request to the UMA oracle. All financial contracts pay a fee to use the oracle.

participate, voters must obtain UMA voting tokens (an ERC20), and their vote is weighted by their balance. If a minimum threshold of voters (by token count) is reached and more than half of them vote for the same value, this value is the resolution of the price request. If neither condition is met, the price remains unresolved until the next round, where it is voted on again.

All successful voters are rewarded, per price request, with freshly minted tokens in proportion to their stake. Non-voters or incorrect voters are implicitly punished through inflation of the token supply.

As described in the whitepaper, the price of the voting token itself is supported by a buyback mechanism, where the oracle fees are used to purchase UMA tokens on the open market whenever the value of corrupting the oracle is greater than the cost of obtaining majority voting power. This should undermine the incentive to corrupt the oracle.

Privileged roles

There are a number of privileged roles that exercise wide-ranging powers over the protocol.

Most importantly, the Risk Labs Foundation controls the wallet that can withdraw oracle fees. They intend to use these fees for the buyback mechanism but this is not currently programmatically enforced. The intention is to encapsulate the buyback logic in a smart contract but until then, token holders must trust the Risk Labs Foundation to perform this function quickly and correctly.

Additionally, there is a `Governor` contract that manages the system in a number of ways:

- It can shutdown or remargin any contract within the system. This is intended as an emergency safety mechanism.
- It can replace the implementation of crucial contracts within the system.
- It can decide which prices are supported by the oracle.
- It can decide which addresses can register new financial contracts (implicitly deciding which financial contracts are supported).
- It can set the oracle fees.
- It can replace the address that can withdraw oracle fees.
- It can set the inflation rate per vote.

Any action that can be undertaken by the `Governor` contract must first be proposed by the Risk Labs Foundation and then ratified by the token holders using the oracle voting mechanism. This has two interesting implications:

- Token holders cannot use the smart contract to initiate an action of their own accord, but they have veto power over any action. The Risk Labs Foundation intends to solicit proposals using an external mechanism, and advance those that have greater than 5% token holder support.
- Since the `Governor` contract relies on the correct functioning of the UMA oracle, it may not be able to correct a problem with the oracle itself.

Ecosystem dependencies

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

Fortunately, this is quite straightforward since the oracle does not interact with external projects. However, it does use time-based logic, which means it is dependent on the availability of Ethereum in multiple ways:

- After a price is resolved, voters have a time limit to collect their rewards. If they are temporarily unable to submit this transaction (for instance, during a period of high Ethereum congestion), the rewards will be lost.
- Similarly, if a voter is unable to commit or reveal within a particular round’s time window, those votes will not be counted.
- Votes that were not included in the round they were made may end up being confirmed in subsequent rounds. In such cases, the voter would typically choose to do that anyway.

Critical severity

None.

The Data Verification Mechanism uses a commit-reveal scheme to hide votes during the voting period. The intention is to prevent voters from simply voting with the majority. However, the current design allows voters to blindly copy each other's submissions, which undermines this goal.

In particular, each commitment is a masked hash of the claimed price, but is not cryptographically tied to the voter. This means that anyone can copy the commitment of a target voter (for instance, someone with a large balance) and submit it as their own. When the target voter reveals their salt and price, the copycat can "reveal" the same values. Moreover, if another voter recognizes this has occurred during the commitment phase, they can also change their commitment to the same value, which may become an alternate Schelling point.

Consider including the voter address within the commitment to prevent votes from being duplicated. Additionally, as a matter of good practice, consider including the relevant timestamp, price identifier and round ID as well to limit the applicability (and reusability) of a commitment.

Update: *Fixed in PR#1217. The hash now includes all relevant context to avoid duplication of votes and limit their applicability and reusability.*

[H02] Governance actions that require ETH cannot be practically executed

The `Governor` contract allows the account with the `Proposer` role to propose new governance actions through the `propose` function. Each proposal is made up of a number of actions, each action represented with the `Transaction` struct. An action is essentially an external call from the `Governor` contract to an address, passing arbitrary data and value.

Actions that require Ether for their execution (i.e., its `value` field is greater than zero) may be difficult to execute and produce unexpected failures since there is no straightforward way to deposit ETH into the `Governor` contract. None of the functions implemented by the contract (nor its ancestors) are marked as `payable`. It must also be noted that while several scenarios are tested in the `Governor.js` file, the case where a proposal involves a transfer of ETH is not covered.

Consider implementing the necessary functionality so that the `Governor` contract can receive ETH before executing a proposal. This should allow actions that require Ether to be easily

`Governor` contract when each transaction within a proposal is executed (via the `executeProposal` function) and not when it is proposed (via the `propose` function). This means that an approved proposal that requires ETH might never be executed if no actor is willing to pay for it.

[H03] Any governance action could be executed multiple times

The `executeProposal` function of the `Governor` contract can execute a transaction listed in an approved proposal. A proposal can contain multiple transactions, each of them expected to be executed a single time in the order they appear listed in the `Proposal.transactions` array.

Once a transaction is executed successfully, the `executeProposal` function deletes the transaction from the proposal so that it cannot be executed twice. However, the removal is done *after* the external call is made, which fails to follow the Checks-Effects-Interactions pattern. This can be leveraged by any contract called by the `Governor` to execute a reentrancy attack, where the callee would reenter the `executeProposal` function and trigger the same transaction multiple times.

It must be noted that while the reentrancy attack is indeed feasible, the likelihood of it actually occurring and having a negative impact in the UMA protocol is diminished by fact that:

- A proposal can only be proposed by the account with the trusted `Proposer` role, which will decide the transactions included in the proposal. It is unlikely that the `Proposer` constructs a malicious transaction that invokes a contract that does a reentrancy.
- Even if the `Proposer` does include a malicious proposal, it would have to be voted by the Governance system before it can be executed. This process should involve reviewing and validating that the proposal does not contain unexpected, potentially malicious transactions that may harm the system.

To ensure a proposal's transaction cannot be executed multiple times leveraging a reentrancy attack, consider modifying the `executeProposal` function of the `Governor` contract to tightly follow the Checks-Effects-Interactions pattern. In particular, the removal of the executed transaction must occur *before* the external call.

Medium severity

[M01] Token holders can react to revealed vote

In each voting round the token balances are snapshotted during the first reveal transaction. If this transaction is front-run, it allows voters to buy or sell tokens based on the contents of the first revealed vote. They may use this to change the stake associated with their vote when they discover whether it matches another voter.

Consider introducing a dedicated transaction to snapshot token balances at the start of the reveal phase so that they can be locked before the first reveal.

Update: *Fixed in PR#1238. The risk of the front-running described in the issue still exists, but a new function `snapshotCurrentRound` has been added to give users an alternative way of freezing the relevant variables before the first reveal in an isolated transaction. The UMA team is aware of the tradeoff between security and user-experience for this particular case. In their words:*

> This is certainly possible, but unlikely due to the difficulty in doing it correctly and the minimal reward for doing so. For that reason, it seemed making the transaction optional was a reasonable compromise between UX and security. If this were to become a problem, bots could be created to call the snapshot transaction as early as possible in the reveal period.

[M02] Fragile implementation of conversion from integer to UTF-8 representation

The private `_uintToBytes` function of the `Governor` contract intends to convert any given integer number into its UTF-8 bytes representation, returned as a `bytes32` type. Even though the function works as intended, we consider its implementation to be dangerously fragile for the following reasons:

- Misleading function name: the function does not return a `bytes` type, but rather a `bytes32`.
- Unclear intention: the fact that the function attempts to build a UTF-8 representation of the passed integer is not clear at all without thorough manual inspection.

- Lack of documentation: the function operates at a byte level to reproduce the UTF-8 representation of an integer, but none of the low-level steps are explained.
- Hardcoded values: the function uses multiple hardcoded values without explaining what they represent.
- Failing silently: the function implicitly discards any digits after the 32nd, failing silently at representing large numbers and producing output collisions.

Consider refactoring, documenting and testing the `_uintToBytes` function of the `Governor` contract to address these shortcomings.

Update: *Fixed in PR#1204.*

[M03] Discrepancies with the whitepaper

There are some discrepancies between the code and the whitepaper. Some (and possibly all) of these are known and acknowledged by the UMA team but we list them for completeness.

1. The whitepaper claims that price requests must be for a datetime within the last two voting periods. In fact, they are restricted to any time in the past.
2. The whitepaper claims the balances are snapshotted at the beginning of a commitment voting period. In fact, they are snapshotted when the first voter reveals their hash.
3. The whitepaper claims that when more than 50% of voters cannot agree, the median is returned as the result. In fact, the vote is simply delayed until the next round.
4. The whitepaper claims that accurate voters increase their token ownership by a fixed factor `r` every voting period. In fact, the increase occurs for every vote.
5. The whitepaper describes a mechanism to prevent parasitic contracts from using the oracle without reporting the potential profit from corruption or paying the fees. This mechanism is not in use. The whitepaper does not actually claim that the UMA DVM uses this mechanism but we believe that this fact is worth clarifying. It should also be noted that the system currently has an emergency shutdown mechanism that could be used to nullify any registered contract (and implicitly, the parasitic contracts that depend on its price requests). However, this mechanism does not distinguish between registered contracts that are intentionally created to be parasitized, and legitimate contracts that use popular price feeds,

Update: *All points acknowledged. Regarding points (1), (2) and (4), the whitepaper will be updated to reflect the implementation. Point (3) will be implemented in the future, and the whitepaper will be updated to clarify. As of point (5), UMA does not currently intend to implement this mechanism, and the whitepaper will be updated to make it explicit.*

[M04] Financial contract registration with repeated parties might lead to inconsistent behaviors

The `registerContract` function of the `Registry` contract takes an array `parties` containing a list of addresses. However, the function never validates whether the passed array contains the same party address multiple times. This means that the `for` loop in lines 96 to 100 can register the same contract address multiple times for a party in its corresponding `partyMap[parties[i]].contracts` array (see line 97), though only registering the index of the *last* added contract address in the `partyMap[parties[i]].contractIndex` mapping (see line 99).

The described behavior can be leveraged to take the `Registry` contract to an inconsistent state, where if multiple repeated parties appear in financial contracts, it may become impossible to fully remove a registered party. Refer to this unit test for a step-by-step reproduction of the issue. Note that in the showcased example, the `Register` contract ends up in an inconsistent state where:

1. When querying the registered contracts for a party, a contract `contract1` appears as registered.
2. When querying the `isPartyMemberOfContract` for the same party and contract `contract1`, the party *does not* appear as a member of the contract.
3. When trying to remove the party from the contract, the transaction is reverted, even if the party has that contract registered as shown in (1).

Financial contract registration in the `Registry` is restricted to known accounts with the `ContractCreator` role, so it would not be possible for everyone to leverage the described behavior maliciously. Nevertheless, the system's behavior might be affected if repeated parties are registered accidentally.

`PartyAdded` event is emitted for the initial parties as well. Afterwards, related unit tests should be added to ensure the system behaves as intended.

Update: *Fixed in PR#1194. An internal `_addPartyToContract` function has been added, which is called when a contract is registered and when a new party is added to an existing contract. Related unit tests have been added as well.*

Low severity

[L01] Possibly miscalculated late penalty

In the `computeRegularFee` function of the `Store` contract, the late penalty fee is calculated using the duration of time that the contract is paying for. It does not include any notion of the current time or the due date. This only makes sense if `startTime` is the time that the fee started accruing and `endTime` is the current time. If this is a requirement, it should be specified and enforced. Otherwise, contracts that use this function to calculate fees for different periods of time may be misinformed about the late penalty.

Depending on how the function is intended to be used, consider replacing the `endTime` parameter with the current block time, or moving the late penalty fee calculation into its own separate function, and documenting the intended use either way.

Update: *Fixed in PR#1237. The late penalty is now scaled by the difference between `startTime` and the current time. Importantly, in the expected use case, it now scales quadratically with the number of unpaid weeks. The regular fee is unchanged.*

[L02] Externally-owned accounts can be registered as contracts in Finder contract

The `changeImplementationAddress` function of the `Finder` contract is a privileged function (only called by the `owner` address) in charge of registering contract addresses under a given interface name. However, while the function is only intended to register contract addresses, there is currently no validation on whether the address provided is actually a contract.

Update: *The issue is acknowledged, but the implementation will remain unchanged. Addressing the issue would introduce an unnecessary burden on existing token holders to migrate to a new contract.*

[L03] Roles cannot be renounced

The role management scheme implemented in the `MultiRole` contract does not include a way for accounts to renounce the roles they are granted. This might become problematic in a scenario where an account wishes to renounce a role after the trusted device holding the private keys has been compromised.

Consider allowing accounts to lose their granted privileges by renouncing any of their roles. For reference, follow the development of the `AccessControl` contract in the OpenZeppelin Contracts library.

Update: *Fixed in PR#1247. Users can renounce their shared roles. Users are intentionally prevented from renouncing exclusive roles as a safety mechanism.*

[L04] Missing error messages in require statements

There are several `require` statements without error messages. For examples, see line 29 in `Withdrawable.sol`, and lines 56, 70, 71 and 125 in `Store.sol`.

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

Update: *Fixed in PR#1259. The `require` statements now contain error messages.*

[L05] Lookup key strings are not centrally defined

Known UMA contracts are tracked in the `interfacesImplemented` mapping of the `Finder` contract. New entries can be added by a privileged address via the `changeImplementationAddress` function, and the `getImplementationAddress` function acts as a public getter to query the registry providing a string-type key. While this registry is used by several different contracts to get the addresses of legitimate UMA contracts, the strings

This issue does not pose a security risk, but the approach taken is error-prone and difficult to maintain. Therefore, consider factoring out all mentioned constant strings to a single library, which can be then imported in the necessary contracts. This will ease maintenance and make the code more resilient to future changes.

Update: *Fixed in PR#1241 and PR#1320. The code base uses the new `OracleInterfaces` library to reference contracts within the system.*

[L06] Use of arbitrary numbers as role identifiers is prone to error

The `MultiRole` contract uniquely identifies registered roles using a `uint` data type. Contracts inheriting from `MultiRole` are then expected to define specific roles and associate them with a number. For example, the `ExpandedERC20` contract defines a `Roles` enum where the `Owner` element would be identified with 0, `Minter` with 1 and `Burner` with 2.

This approach might be error-prone when dealing with multiple contracts implementing different roles. For instance, the role allowed to withdraw from a `Withdrawable` contract might also be identified with 0, thus semantically overloading the identifier `0`, which would mean different things depending on which contract it references.

Consider uniquely labeling roles with identifiers that are more self-explanatory and less prone to confusion. In particular, consider using hashes of meaningful strings as role identifiers (e.g., `keccak256("MINTER_ROLE")`). For reference, follow the development of the `AccessControl` contract in the OpenZeppelin Contracts library.

Update: *Acknowledged, but the implementation will remain unchanged. Addressing this issue would necessitate a major change to the `MultiRole` interface.*

[L07] Not failing loudly

The `addSupportedIdentifier` and `removeSupportedIdentifier` functions of the `IdentifierWhitelist` contract do not revert when the expected state changes do not take place.

Update: *Acknowledged, but the implementation will remain unchanged. The suggested mitigation introduces an additional risk, since the `Governor` contract adds and removes a price identifier and an unexpected failure may disable it.*

[L08] Lack of input validation

In the `Store` contract:

- The `setWeeklyDelayFee` function does not validate whether the `newWeeklyDelayFee` parameter is less than 1.

In the `VoteTiming` library:

- The `init` function does not validate that the `phaseLength` parameter is greater than zero. This is important to avoid unexpected divisions by zero in other functions of the library.

In the `Shared` library:

- The `addMember` function does not do a null address check like the `resetMember` function of the `Exclusive` library.

Consider implementing the necessary logic where appropriate to validate all relevant user-controlled input.

Update: *Fixed in PR#1212. The suggested validations are implemented.*

[L09] Repeated reward events

In the `retrieveRewards` function, a `RewardRetrieved` event with zero tokens is emitted when the reward is expired or incorrect. Additionally, after a correct vote is processed and cleared, if it is processed again it will fail the validity test and be treated as an incorrect vote. In any of these situations, the same vote can be processed multiple times, emitting a redundant `RewardRetrieved` event every time.

Following the “fail early” principle, consider checking that the `revealHash` is non-zero before processing a vote. This should help prevent emitting the `RewardRetrieved` event

[L10] Transfer of ETH may unexpectedly fail

The `withdraw` function of the `Withdrawable` contract allows authorized callers to withdraw any amount of ETH from the contract. The transfer of Ether is executed with Solidity's `transfer` function, which forwards a limited amount of gas to the receiver. Should the receiver be a contract with a fallback function that needs more than 2300 units of gas to execute, the transfer of Ether would inevitably fail. After the Istanbul hard fork, it has become a recommended practice not to use `transfer` to avoid hard dependencies on specific gas costs.

To avoid unexpected failures in the withdrawal of Ether, consider replacing `transfer` with the `sendValue` function available in the OpenZeppelin Contracts library.

Update: *Fixed in PR#1225.*

[L11] Ever-growing array of pending price requests

The `Voting` contract tracks all price requests still to be resolved in its `pendingPriceRequests` array. Once a price request is considered resolved, the `_resolvePriceRequest` function attempts to remove the corresponding request from `pendingPriceRequests`. This is done by first replacing the request with the last item in the array and then updating the associated `priceRequests` mapping to keep indices updated and consistent. However, the function falls short at fully cleaning the `pendingPriceRequests` array. In particular, while the elements are properly deleted, the array's length is never decreased.

It must be noted that an ever-growing `pendingPriceRequests` array does not cause any security issues. Yet the array will be unnecessarily filled with zero elements, which would have a negative impact on the `getPendingPriceRequests` function's gas efficiency.

Consider decreasing the length of the `pendingPriceRequests` array every time an element is deleted. Furthermore, consider adding related unit tests to ensure this unexpected behavior is not reintroduced in future modification to the code base.

Update: *Fixed in PR#1207.*

[L12] Erroneous comments

associated Party struct”.
- In line 76 of `Registry.sol`, the comment is an incomplete thought.
- In line 127 of `Registry.sol`, “to” should be “from”.
- In line 25 of `VoteTiming.sol`, “floor(timestamp/phaseLength)” should say “floor(timestamp/roundLength)”.
- In line 22 of `DesignatedVoting.sol`, an inline comment claims that the `Owner` role can set the `Withdrawer` role, but actually the `Owner` and `Withdrawer` roles are permanently the same role.
- In line 55 of `ResultComputation.sol`, the `minVoteThreshold` is described as an inclusive bound but it is actually an exclusive bound. Similarly, the comment in line 72 should say “exceeded” instead of “met”.
- In line 549 of `Voting.sol`, the comment references a non-existent `isActive` function.
- In line 71 of `Governor.sol`, there is a redundant `@param` comment.
- In line 154 of `Governor.sol`, the comment states the proposal data will be zeroed after execution but the `requestTime` will remain.

Update: *Partially fixed in PR#1213. The comment in line 127 of `Registry.sol` has been fixed in PR#1206. The erroneous comment in line 25 of `VoteTiming.sol` has not been addressed.*

[L13] Nonexistent role can be set in Withdrawable contract

The internal `setWithdrawRole` function of the `Withdrawable` contract can be used by derived contracts to set the role identifier allowed to withdraw Ether or tokens. However, the function does not validate whether the passed `roleId` argument is already registered. While the function’s docstrings explicitly state that the role must exist, this should be checked programmatically to avoid errors.

Consider using the available `onlyValidRole` modifier to effectively validate that the `roleId` argument exists.

Update: *Fixed in PR#1226.*

[L14] Lack of event emission after fee update

UMA system, consider logging an event after potentially sensitive modifications are applied.

Update: *Fixed in PR#1214.*

[L15] Actions in proposals might fail silently when target is an externally-owned account

The `executeProposal` function of the `Governor` contract internally calls the private `_executeCall` function to execute the external call involved in a proposal's action. In turn, `_executeCall` uses the low-level `call` function to pass arbitrary data to the target address. However, the function never validates whether the address being called is actually a contract.

The intended use case of the `Governor` contract is to manage the UMA system contracts so proposals will typically include transactions to those contracts, and will be vetted by the UMA DVM. Nevertheless, any assumptions should be represented by explicit constraints in the code. If the transaction contains data but the target is an externally owned account, the external call will still be considered successful (i.e., `_executeCall` will return `true`) even though no code was executed.

Consider validating proposals during registration to ensure that, if any data is included in an action, the target address is indeed a deployed contract.

Update: *Fixed in PR#1242. Proposals are rejected if they include transactions that would send data to an externally owned account.*

[L16] Unsafe addition in reward's expiration time calculation

The `Voting` contract calculates the reward's expiration time using an unsafe addition. In practice, this is unlikely to overflow. Nevertheless, consider using OpenZeppelin's `SafeMath` library for all integer calculations.

Update: *Fixed in PR#1235.*

[L17] Immutable designated voting association

The `DesignatedVotingFactory` contract is designed to let a voting address (typically a hot wallet) register an associated `DesignatedVoting` contract that lets them vote on behalf of an

While this does not pose a security issue, it does prevent the mapping from tracking changes to the `DesignatedVoting` contract, which may degrade user experience.

Consider including a `clearDesignatedVoting` function to remove the association.

Update: *UMA acknowledges this issue, but the implementation will remain unchanged. According to UMA, addressing it would introduce an unnecessary burden on existing token holders to migrate to a new contract.*

[L18] Using unstable version of OpenZeppelin Contracts may break functionality

The OpenZeppelin Contracts library is imported as `"@openzeppelin/contracts": "^3.0.0-beta.0"`. This is an unstable, pre-production version of the library, which may include unexpected breaking changes between releases until it is stabilized. For example, since the dependency is not pinned, the latest `3.0.0-rc.0` version is now installed instead of the `3.0.0-beta.0` version. This first release candidate includes breaking changes from the previous beta version, such as contracts moved to other folders (e.g., `Ownable`, or `ERC20Snapshot`), or functions changing names and visibility. Some of these changes have broken imports in the UMA project and contracts now fail to compile correctly. Furthermore, the previously public `snapshot` function of the `ERC20Snapshot` contract has been restricted to `internal`, and its name has changed to `_snapshot`.

Initially, consider pinning the imported dependency to a fixed version to avoid installing undesired versions that may break functionality. More importantly, before moving into production, consider waiting for the final stable release of OpenZeppelin Contracts 3.0.

Update: *Fixed in PR#1254 and PR#1280. The project is now using OpenZeppelin Contracts v3.0.0*.

[L19] Redundant and fragile ERC20 transfer

The `payOracleFeesErc20` function of the `Store` contract allows the caller to deposit ERC20 tokens into the `Store` contract. As the transfer of tokens is executed via the ERC20 `transferFrom` function, the caller must have previously approved the tokens to the `Store`

This implementation can be considered suboptimal for a few reasons. Firstly, the approve-then-pull pattern is typically used as a workaround so that contracts can respond to receiving tokens. Since the `payOracleFeesErc20` function does not emit any events or perform any internal actions, the workaround is unnecessary. Secondly, the allowance is not typically considered to be equivalent to a payment. It is common practice to assign a high allowance and allow contracts to retrieve the required amount when necessary. Lastly, there are some ERC20 tokens that use an allowance of `2 ** 256 - 1` as a flag to indicate “infinite” allowance. Two well-known examples include the DAI token and Compound Finance’s CTokens. In such a case, `payOracleFeesErc20` will be unable to retrieve tokens when granted an “infinite” allowance.

Consider removing the `payOracleFeesErc20` function in favor of direct ERC20 transfers. Alternatively, consider letting the caller determine how many tokens are to be transferred from their accounts to the `Store` contract.

Update: *Fixed in PR#1255. The `payOracleFeesErc20` function explicitly accepts the amount to transfer.*

[L20] Tests not passing

The testing suite finishes with 4 failing tests. These tests are:

```
1) Contract: scripts/Voting.js
Intrinio price:

AssertionError: expected 0 to equal 1
+ expected - actual

-0
+1

at Context. (test/scripts/Voting.js:248:12)
at processTicksAndRejections (internal/process/next_tick.js:81:5)
```

```
AssertionError: expected 1 to equal 0
+ expected - actual

-1
+0

at Context. (test/scripts/Voting.js:316:12)
at processTicksAndRejections (internal/process/next_tick.js:81:5)

3) Contract: scripts/Voting.js
Numerator/Denominator:

AssertionError: expected 2 to equal 1
+ expected - actual

-2
+1

at Context. (test/scripts/Voting.js:352:12)
at processTicksAndRejections (internal/process/next_tick.js:81:5)

4) Contract: scripts/Voting.js
Only batches up to the maximum number of commits or reveals that ca

There should be 0 pending requests during pre-commit phase
+ expected - actual

-3
+0

at Context. (test/scripts/Voting.js:380:12)
at processTicksAndRejections (internal/process/next_tick.js:81:5)
```

As the test suite was left outside of the audit's scope, please consider thoroughly reviewing the test suite to make sure all tests run successfully. Furthermore, it is advisable to only merge code that

Update: *Not an issue. As UMA correctly pointed out, these tests do pass if the environment variables are set correctly.*

[L21] Unvetted function call

The `withdrawErc20` function in the `Withdrawable` contract and the `payOracleFeesErc20` function in the `Store` contract both call an unvetted function (`transfer` and `transferFrom` respectively) on a user-specified address. If the address is chosen maliciously, this could result in arbitrary code being executed within another contract on behalf of the the UMA contract. This does not directly affect the UMA system but it does introduce an unexpected behavior. To improve predictability, consider whitelisting or vetting the contracts that can be passed to the `withdrawErc20` and `payOracleFeesErc20` functions.

Update: *This issue is acknowledged because, according to UMA, the suggested mitigation would introduce too much complexity for minimal gain.*

[L22] Use of uninitialized state variables in fee calculation

The `computeRegularFee` function of the `Store` contract is in charge of computing the regular oracle fees that a contract should pay, given a period of time in seconds and the profit from corruption. The calculation depends on two state variables `fixedOracleFeePerSecond` and `weeklyDelayFee`, which should be initialized by the owner of the `Store` contract calling the `setFixedOracleFeePerSecond` and `setWeeklyDelayFee` functions respectively. However, these two state variables are not initialized upon construction of the `Store` contract, and the `computeRegularFee` function does not validate whether they are already initialized. As a result, all regular fees calculated *before* the owner sets the `fixedOracleFeePerSecond` and `weeklyDelayFee` state variables will inevitably be zero.

To avoid unexpected behaviors, consider initializing the `fixedOracleFeePerSecond` and `weeklyDelayFee` state variables during construction of the `Store` contract.

Update: *Fixed in PR#1256. The variables are set in the constructor of the `Store` contract.*

Notes & Additional Information

`VoteTiming` library, the round ID depends on the global timestamp but not on the lifetime of the system. Although this is a reasonable choice to simplify the implementation, it has the surprising consequence that the initial round ID starts at an arbitrary number (that increments, as expected, for subsequent rounds) instead of zero or one. To favor readability, consider stating this behavior explicitly in the function's comments.

Update: *Fixed in PR#1272.*

[N02] 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 example line 65 of `ResultComputation.sol`, line 7 of `ContractCreator.sol` or line 194 of `FixedPoint.sol`.

Update: *Fixed in PR#1250.*

[N03] Use of uint type

Several variables are declared as `uint` throughout the code base. To favor explicitness, consider changing all instances of `uint` to `uint256`.

Update: *Fixed in PR#1230.*

[N04] Typographical errors

- In `IdentifierWhitelist.sol`:

- Line 30 should say "will succeed" instead of "will be succeed".

- Lines 31, 45 and
  62 have the same unclear parameter description.

- In `IdentifierWhitelistInterface.sol`:

- Line 12 should say "will succeed" instead of "will be succeed".

- Line 190 should say “initialMember” instead of “initialMembers”.

- In `OracleInterface.sol`:

- Lines 13, 21 and 30 contain the extra word “of”.
- Line 31 misspells “identifier”.

- In `Registry.sol`:

- Line 127 should say “from the calling contract” instead of “to the calling contract”.

- In `ResultComputation.sol`:

- Line 55 has the extra word “been”.
- Line 95 has the extra word “correctly”.

- In `Voting.sol`:

- Line 215 has an extra word “of”.
- Line 279 says “time” instead of “type”.
- Line 279 should say “an identifier” instead of “a identifier”.
- Line 280 misspells the word “list”.
- Lines 314 and 346 have the extra word “is”.
- Line 396 references `EncryptedSender.sol` but there is no such file in the repository.
- Lines 605 and 614 should say “round’s” instead of “rounds”.
- Line 711 misspells the word “snapshotted”.

- In `VotingInterface.sol`:

- Line 46 has the extra word “is”.
- Line 47 describes the salt parameter in a confusing way.
- Line 56 should say “array of structs” instead of “struct”.
- Line 106 describes a named return parameter (`totalRewardToIssue`) but the name is not part of the function signature.

The ERC20 `transfer` and `transferFrom` operations executed in line 29 of `Withdrawable.sol` and line 71 of `Store.sol` respectively are correctly wrapped in `require` statements to handle ERC20 contracts that return `false` upon failure. Yet, in other parts of the code base (out of this audit's scope), it was noticed that OpenZeppelin's `SafeERC20` library is used for ERC20 operations.

To be consistent throughout the code base and avoid confusion, consider modifying the `Withdrawable` and `Store` contracts to use the `SafeERC20` library.

Update: *Fixed in PR#1205.*

[N06] Inconsistencies in coding style

Minor deviations from the Solidity Style Guide were found. For example, internal functions in lines 37 and 47 of `Withdrawable.sol` should start with an underscore. Additionally, while some functions parameters start with an underscore, others do not.

Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style with help of linter tools such as Solhint is recommended.

Update: *Fixed in PR#1271. The related `STYLE.md` file has been updated accordingly in PR#1286.*

[N07] Naming issues

In the `MultiRole` contract:

- The `resetMember` function should be renamed to `resetExclusiveMember`.
- The `getMember` function should be renamed to `getExclusiveMember`.
- The `addMember` function should be renamed to `addSharedMember`.
- The `removeMember` function should be renamed to `removeSharedMember`.

In the `Withdrawable` contract:

In the `Store` contract:

- The `computeFinalFee` function should be renamed as it does not actually compute anything. If keeping the name for consistency, this should be explicitly stated in docstrings to avoid confusions.
- The `fixedOracleFeePerSecond` and `weeklyDelayFee` variables should be renamed to `fixedOracleFeePerSecondPerPFC` and `weeklyDelayFeePerPFC` or similar.

In the `EncryptedStore` contract:

- The contract name `EncryptedStore` is misleading, as it does not actually enforce any kind of encryption. It should be renamed to `MessageStore` or similar.

In the `Governor` contract:

- The `_uintToBytes` function should be renamed to `_uintToBytes32`.

In the `Voting` contract:

- The `getPendingRequests` function should be renamed to `getActiveRequests` or similar, since it only returns active price requests. In particular, it does not return requests scheduled to be resolved in a future round.

In the `FixedPoint` contract:

- The `divRaw` variable should be renamed to `aScaled` or similar, since it has nothing to do with division.

Update: *Fixed in PR#1227, PR#1231 and PR#1204. The `MultiRole` library will remain unchanged, as well as the `getPendingRequests` function of the `Voting` contract.*

[N08] Potentially differing time sources in testing environment

The `Testable` contract is a base contract from which all contracts that need to mock time changes in testnets should inherit. If several contracts of the UMA protocol derive from the implementing a pattern that keeps a single source of time in testing environments.

Update: *Fixed in PR#1236. All contracts use a shared `Timer` contract during testing to obtain the mocked time.*

[N09] Named return variables

Named return variables are used inconsistently throughout the code base. For example, the `computeCurrentRoundId` and `computeRoundEndTime` functions of the `VoteTiming` library have named return variables in the function signature but return their results directly.

Consider removing all named return variables, explicitly declaring them as local variables, and adding the necessary `return` statements where appropriate. This should improve both explicitness and readability of the project.

Update: *Fixed in PR#1229. Named returns are only consistently used in functions that return multiple values.*

[N10] Consider warning voters about salt reuse

When voters reveal their previously committed vote using the `revealVote` function of the `Voting` contract, they must provide the salt used to mask the vote hash. Since transaction data is public, the salt will be revealed with the vote. While this is the system's expected behavior, it assumes that voters will never reuse the same pair of salt and price. Otherwise their commits could be easily disclosed in advance.

Since this is a common mistake, consider adding user-friendly documentation, both in docstrings and external documentation, stating the risks of reusing salts in the commit and reveal voting scheme.

Update: *Fixed in PR#1273.*

[N11] Hardcoded condition in require statement

To favor simplicity, consider replacing the `require` statement in line 114 of `MultiRole.sol` with a `revert` statement.

The code base uses the Ethereum Natural Specification (NatSpec) format inconsistently. In particular the `@param` and `@return` tags are often missing. Consider adding the missing tags to all contracts and functions.

Update: *Fixed in PR#1270.*

[N13] Redundant inheritance from MultiRole contract

Contracts `DesignatedVoting` and `Store` inherit from both `MultiRole` and `Withdrawable` contracts. However, `Withdrawable` already inherits from `MultiRole`. To favor simplicity and avoid confusion, consider removing the redundant inheritance from `MultiRole` in the `DesignatedVoting` and `Store` contracts.

Update: *Fixed in PR#1203. It should be noted that the related import statements were not removed.*

[N14] Untested, undocumented behavior of late penalty fee

The `computeRegularFee` function of the `Store` contract computes a penalty fee that is to be paid if the regular fee is overdue. The penalty fee is expected to be computed per overdue week and currently is always floored. For example, if the regular fee is more than a week overdue but less than two, the penalty fee will be computed as if a single week had passed. While this appears to be the system's expected behavior, it was found to be undocumented and barely tested, which may cause confusions in developers, auditors and users alike.

Consider explicitly documenting the behavior of the penalty fee and adding relevant unit tests to ensure the system behaves as intended.

Update: *Fixed in PR#1251.*

[N15] Unused encrypted vote

When a voter commits to a price, they can optionally include an encrypted version of their vote to be stored in the `Voting` contract. This value is never decrypted, verified against their hash or processed in any way within the EVM. Instead, it simply uses the EVM as a temporary storage efficient than modifying EVM storage.

Update: *Fixed in PR#1231. The `EncryptedStore` contract has been removed, and encrypted votes are logged using the `EncryptedVote` event of the `Voting` contract.*

[N16] Base contracts not marked as abstract

All base contracts that are not intended to be instantiated directly, such as `Withdrawable`, `MultiRole`, `Testable` or `ContractCreator` should be marked as `abstract` to favor readability and avoid unintended usage.

Update: *Fixed in PR#1201.*

[N17] Implicitly merged withdraw roles

In the `withdraw` and `withdrawErc20` functions of the `Withdrawable` contract, the caller is also the recipient of the funds. For increased flexibility, consider allowing the withdrawer to specify the recipient address.

Update: *The UMA team has decided not to move forward with our recommendation*.

Conclusion

Originally, no critical and three 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**.

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