OpenZeppelin

UMA Optimistic Asserter Audit

OPENZEPPELIN SECURITY | JANUARY 30, 2023

Security Audits

January 10, 2023

This security assessment was prepared by **OpenZeppelin**.

Table of Contents

Summary

Type	DeFi
Timeline	From 2022-12-05 To 2022-12-09
Languages	Solidity
Total Issues	21 (16 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	1 (0 resolved)
Low Severity Issues	5 (4 resolved)
Notes & Additional Information	

We audited the UMAprotocol/protocol repository at the `fed1c5677b5e764eb0b3e59817941c129e6c26f7` commit.

In scope were the following contracts:

```
packages/core/contracts/optimistic-asserter
├── implementation
│   ├── OptimisticAsserter.sol
│   └── escalation-manager
│       ├── FullPolicyEscalationManager.sol
│       └── BaseEscalationManager.sol
└── interfaces
    ├── OptimisticAsserterCallbackRecipientInterface.sol
    ├── OptimisticAsserterInterface.sol
    └── EscalationManagerInterface.sol
```

System Overview

UMA's `OptimisticAsserter` contract uses the same incentive mechanism as the Optimistic Oracle to quickly resolve claims that could be resolved by the (slow) Data Verification Mechanism (DVM). The main difference is that rather than being responses to previous requests, claims are asserted directly.

In both cases, the claim asserter posts an associated bond, and there is a short time window where someone else can dispute the claim and post their own bond. Disputes are adjudicated by the existing DVM and the loser's bond is sacrificed and partially transferred to the winner. If the DVM is functioning correctly, its decision is predictable, so there is no reason to post invalid claims or make invalid disputes. If the claim is not disputed within the time window, it can be assumed to be valid without invoking the DVM.

The `OptimisticAsserter` contract also allows the process for resolving claims to be customized by the claim asserter. Specifically, they can choose:

- Whether to disregard the oracle result if the claim is disputed
- Whether to limit the possible disputers

This allows the contract to be used in a much larger set of contexts. The last three properties are decided by assigning an *Escalation Manager*, which is a contract that can choose these policies on a per-assertion basis.

Even when the “disregard oracle” option is chosen, the dispute still needs to be resolved by the oracle in order to distribute the bond. The flag simply indicates that the asserter will not wait for the resolution and will likely create a new assertion instead.

Security Model and Trust Assumptions

Privileged Roles

The `OptimisticAsserter` contract has an owner that can choose the fraction of the loser’s bond that is sent to the `Store` contract when a dispute is resolved by the DVM. The owner can also set the default bond currency and liveness parameters, which apply whenever the claim asserter chooses to use the default values. Consequently, asserters must implicitly trust the owner not to front-run their assertion with unreasonable parameters.

The `FullPolicyEscalationManager` contract, intended to be used as an escalation manager, also has an owner that has complete control over which assertions it will accept, the oracle results, the disputer whitelist, and whether the oracle results are disregarded.

Assertion Configuration

Given the extensive customization possibilities, there are several ways for a claim asserter to create invalid assertions that will nevertheless not be disputed. For example, they could choose an unreasonably small dispute window, a prohibitively large bond, callbacks that revert dispute operations, or even an escalation manager that provides a biased oracle or refuses valid disputers. Consequently, potential disputers and anyone who relies on a resolved claim should ensure the assertion was fairly structured. In many cases, the relevant participants (claim asserters or escalation manager owners) will be contracts that can be vetted for reasonable configurations.

Medium Severity

Burn percentage applies retroactively

When creating an assertion, the minimum bond depends on the current value of the configurable burn percentage. Asserters and disputers would reasonably expect their reward to be the bond reduced by the `burnedBondPercentage` at the time the assertion was originally made. However, in contrast to other configurable parameters, the burn percentage is not cached, resulting in the actual reward being reduced by the most recent value at the time of settlement.

Consider caching the burn percentage and using it consistently throughout the life of an assertion.

Update: *Acknowledged, not resolved. The UMA team stated:*

> *We've decided to not make this change. The implications of the issue presented here is that the voters vote to change the bond burn percentage after an assertion is disputed with the goal of stealing it from the disputer (for example setting the bond burn percentage to 100% after an assertion is made). We feel that this is a non-issue as the reputational damage of the DVM doing this would destroy the project and the cost to the protocol would be more than the value extracted from taking the reward.*
>
> *Additionally, we are at the stack depth for the assertion object and adding an additional property to an assertion (bond burn percentage at assertion time) would require additional data structures or other refactors to accommodate this that we don't want to do at this time.*

Low Severity

Arbitration resolution can be modified

The `setArbitrationResolution` function in the `FullPolicyEscalationManager` contract can be called multiple times for the same `requestId`. This would allow the contract owner to change the price for a given `requestId`, invalidating the `resolvedPrice` result in

Consider restricting the `setArbitrationResolution` function to only allow setting the resolution once per `requestId`.

Update: *Resolved in pull request #4319 with commit 76bed2a.*

Lack of access control on potentially sensitive functions

In the `BaseEscalationManager` contract, the following `public` functions may benefit from access control:

- `requestPrice`
- `assertionResolvedCallback`
- `assertionDisputedCallback`

Currently anyone can call these functions, which can result in them executing state-changing logic in an unintended context.

Consider implementing appropriate access control on these functions so that only the expected instance(s) of the `OptimisticAsserter` contract can call them.

Update: *Resolved in pull request #4334 with commit dd99434.*

Erroneous docstrings and comments

Consider fixing the following docstrings and inline comments:

- In `OptimisticAsserter.sol`, the comment on line 88 claims the caller is the asserter, but that is not necessarily true.
- In `FullPolicyEscalationManager.sol`, comments on line 161, line 171, and line 180 for the respective `setDisputeCallerInWhitelist`, `setWhitelistedAssertingCallers`, and `setWhitelistedAsserters` functions state “Adds a … to the whitelist”. In reality, the functions can both add and remove from the whitelists.

Update: *Resolved in pull request #4320 with commit b943b23.*

- In BaseEscalationManager.sol:
  - `assertionResolvedCallback` has undocumented `assertionId` and `assertedTruthfully` parameters.
  - `assertionDisputedCallback` has an undocumented `assertionId` parameter.
- In EscalationManagerInterface.sol:
  - The interface is undocumented.
  - All functions and structs are undocumented.
- In FullPolicyEscalationManager.sol:
  - `setDisputeCallerInWhitelist` has an undocumented `value` parameter.
  - `setWhitelistedAssertingCallers` has an undocumented `value` parameter.
  - `setWhitelistedAsserters` has an undocumented `value` parameter.
- In OptimisticAsserter.sol:
  - `assertTruthWithDefaults` has an undocumented `asserter` parameter.
  - `assertTruth` has an undocumented return value.
  - `getCurrentTimestamp` has an undocumented return value.
  - Functions marked `internal` are undocumented.
- In OptimisticAsserterInterface.sol:
  - The interface is undocumented.
  - All functions are undocumented.
  - The structs are only partially documented.
- In OptimisticAsserterCallbackRecipientInterface.sol:
  - The interface is undocumented.
  - All functions are undocumented.

Incomplete documentation hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and facilitate maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned, and the events emitted.

Update: *Resolved in pull request #4342 with commit c831b8c.*

No check for final fee when adding currency to whitelist

When a currency is whitelisted and cached in the `OptimisticAsserter` contract, it is assumed that the `Store` contract will have a final fee set for the currency address. If the final fee has not been set, then assertions can be posted without a bond.

Prior to whitelisting a currency, consider adding a check that ensures the final fee from the `Store` contract is non-zero in the `syncUmaParams` and `_validateAndCacheCurrency` functions.

Update: *Acknowledged, not resolved. The UMA team stated:*

> *We've decided to not make this change. There should never be collateral currencies added that don't have a final fee by DVM. If this was to occur, it would not pose too much of an issue within the system as 0 bonded assertions don't really enable you to do much other than spam the DVM. if you were to spam the DVM with this technique we can delete the requests at the DVM level.*
>
> *Added to this (and more importantly) there is a feature of having zero-sized bonds in one important situation: testnets. It is really convenient to not have to force users on testnets to approve and pay collateral currencies when making simple tests as this strictly increases the friction with using the system.*

Notes & Additional Information

BaseEscalationManager is not declared abstract

The `requestPrice` and `getPrice` functions in the `BaseEscalationManager` contract are stubs that do not perform any oracle actions. Additionally, the `getPrice` function has a return type of `int256` but does not return any value. In its current state, this base contract does not implement a fully functional escalation manager.

***Update:** Acknowledged, not resolved. The UMA team stated:*

> *We think it's better for this to not be abstract as it makes derived contracts easier and shorter to implement; if you don't care about a particular feature then you just do nothing and it defaults to the disabled/simple behavior. For example if you don't care about disabling disputes then the base implementation just returns true for `isDisputeAllowed`. By making this abstract we'd force implementing contracts to have to declare and implement things, even if they don't care about that functionality.*

Constant not using UPPER_CASE format

In `OptimisticAsserter.sol`, the `defaultIdentifier` constant is not declared using `UPPER_CASE` format.

According to the Solidity Style Guide, constants should be named in all capital letters with underscores separating words. For better readability, consider following this convention.

***Update:** Acknowledged, not resolved. The UMA team stated:*

> *While it goes against the solidity styling guide, we'd like to keep this lowercase so that the accessor function `optimisticAsserter.defaultIdentifier()` looks the same as the other assessors of similar type (`optimisticAsserter.defaultCurrency()` and `optimisticAsserter.defaultLiveness()`).*

Escalation managers are bound by DVM constraints

The `OptimisticAsserter` contract is designed to allow disputed assertions to be arbitrated by either the UMA Data Verification Mechanism (DVM) or an escalation manager contract specified by the asserter. The `arbitrateViaEscalationManager` setting controls whether the DVM or escalation manager will act as the oracle for arbitration.

Even when an escalation manager is chosen instead of the DVM, the `assertTruth` function still performs checks that are DVM-specific:

```
require(bond >= getMinimumBond(address(currency)), "Bond amount toc
```

These are restrictions put in place in order to interact with the DVM, but they are not necessarily required by a generic escalation manager. These restrictions force all escalation managers to use the same currencies as the DVM for bond payment, and also impose a minimum bond amount based on the DVM final fee that is unrelated to each escalation manager's costs. The identifiers supported by each escalation manager must also match those supported by the DVM.

Additionally, while the existing design allows the DVM to collect a fee for providing oracle services, it does not allow an escalation manager to collect any fee to cover its potential cost for providing the same service. Moreover, since the usage fee and burn amount are coupled, none of the bond is burned when the escalation manager is used as an oracle.

To provide a more generic and flexible design, consider uncoupling escalation managers from the constraints of the DVM and allowing all oracles to specify their supported identifiers, currencies, and fees. Applying an equal treatment to all oracles should allow for the removal of special-case logic for handling different oracle types.

Update: *Resolved in pull request #4343 with commit 25d22a9. The* `oracleFee` *computed in* `settleAssertion` *is now collected in all dispute cases, even when the escalation manager is used as the oracle, to ensure that there is always a fee associated with any dispute. Additionally, the UMA team stated:*

> *The feature of using escalation manager for dispute arbitration is designed only as a fallback mechanism to allow integrating partners to unplug from UMA DVM if it is considered under risk of being corrupted. This is not meant to be feature-full replacement of existing UMA Oracle system, thus we kept the same DVM constraints also when arbitrating disputes within the escalation manager. Otherwise it would require escalation managers to implement their own whitelisting and fee management mechanisms that is too much overhead just for the fallback scenario.*

Lack of indexed parameters

Within the codebase, some events do not have their parameters indexed. Indexing event parameters facilitates the task of off-chain services searching and filtering for specific events. In

In FullPolicyEscalationManager.sol:

- `DisputeCallerWhitelistSet` event: Consider indexing `address disputeCaller`.
- `AssertingCallerWhitelistSet` event: Consider indexing `address assertingCaller`.
- `AssertingWhitelistSet` event: Consider indexing `address asserter`.

***Update:** Resolved in pull request #4323 with commit 6ad9643.*

Argument name mismatch between interface and implementation

The argument to the `getMinimumBond` function in the `OptimisticAsserterInterface` interface is named `currencyAddress`, but it is named `currency` in the `OptimisticAsserter` contract on line 358.

For clarity, consider renaming one of the arguments so the interface matches the implementation.

***Update:** Resolved in pull request #4324 with commit dffb20e.*

Missing event parameter

In contrast to the `AssertionMade` and `AssertionSettled` events, the `AssertionDisputed` event does not emit the calling address. Consider including it for completeness.

***Update:** Resolved in pull request #4325 with commit 9c37198.*

Lack of SPDX license identifiers

All of the contract files in the `escalation-manager` directory lack SPDX license identifiers.

To avoid legal issues regarding copyright and follow best practices, consider adding SPDX license identifiers to files, as suggested by the Solidity documentation.

***Update:** Resolved in pull request #4326 with commit 5e36c85.*

accept any ETH. In the interest of clarity, consider removing the `payable` keyword.

***Update:** Resolved in pull request #4327 with commit 1d416c9.*

Missing check for non-zero address before calling external function

The `_isDisputeAllowed` function in the `OptimisticAsserter` contract may call `isDisputeAllowed` on an optional `EscalationManagerInterface` contract. In practice, the policy is to not validate disputers when no escalation manager is specified, so the function will exit early instead of invoking a function on the zero address.

Nevertheless, in the interest of local reasoning, consider adding a zero-address check on the `em` address prior to using it to make external function calls. Also consider swapping line 449 and line 450 so that the `em` address assignment is only made when this variable is required.

***Update:** Resolved in pull request #4336 with commit 9543282.*

Redundant code

Consider addressing the following instances of redundant code within the codebase:

- The `override` keyword can be removed from all functions in the `BaseEscalationManager` contract.
- The `requestId` is computed on line 89 and line 155 in the `FullPolicyEscalationManager` contract. Consider deduplicating this code by creating a shared `getRequestId` function. Additionally, consider making this function publicly available for user convenience.
- Within the `FullPolicyEscalationManger` contract, the `require` statement in the `configureEscalationManager` function can be simplified: `require(!_blockByAsserter || (_blockByAsserter && _blockByAssertingCaller), "Cannot block only by asserter");` Execution can only reach the `(_blockByAsserter && _blockByAssertingCaller)` condition if the first condition `!_blockByAsserter` evaluates to false, which means the second condition

of `assertions[assertionId].escalationManagerSettings.escalationManager` happens on lines 435, 443, 449, 471, 472, 480, and 481 in the `OptimisticAsserter` contract. Consider modifying the existing `_getEscalationManager` function to return an `address` type, and using it in all the identified locations other than line 435 to perform the `escalationManager` lookup. This change would also enable the removal of the `address()` cast that occurs on line 414 where `_getEscalationManager` is currently used.

- In the `_callbackOnAssertionResolve` and `_callbackOnAssertionDispute` functions within the `OptimisticAsserter` contract, the lookup of `assertions[assertionId].callbackRecipient` and `assertions[assertionId].escalationManagerSettings.escalationManager` happens twice within each function's scope. In each case, consider storing the value obtained from the first lookup in a local variable for reuse.
- The `return` statement in the `assertTruth` function is redundant because `assertionId` is a named return variable in the function specification. Consider removing this return statement.

Update: *Partially resolved in pull request #4335 with commit 59b13d3. The `_isDisputeAllowed` function still performs the lookup `assertions[assertionId].escalationManagerSettings.escalationManager` instead of using the `_getEscalationManager` function.*

Public functions that could be declared external

In Solidity, functions with public visibility can be used both internally and externally. In the case a function is only ever used externally, it is best practice to limit the visibility to external rather than public. Below is a list of functions that have public visibility that can be limited to external visibility in the `OptimisticAsserter` contract:

- `assertTruthWithDefaults`
- `disputeAssertion`

visibility of these functions to external.

***Update:** Resolved in pull request #4337 with commit bbe05e1.*

Typographical errors

Consider correcting the following typographical errors:

In `BaseEscalationManager.sol`:

- Line 34: “if the dispute” should be “true if the dispute”.
- Line 55: “Optimistic” should be “Optimistic Asserter”.

In `FullPolicyEscalationManager.sol`:

- Line 12: “a whitelistedDisputeCallers” should be “whitelistedDisputeCallers”.
- Line 101: “Assertor” should be “Asserter”.
- Line 161: “disputerCaller” should be “disputeCaller”.

In `OptimisticAsserter.sol`:

- Line 122: “security properties the assertion” should be “security properties for the assertion”.
- Line 128: “Must be a pre-approved.” is an incomplete sentence.
- Line 132: “should be bytes32” should be “0”.
- Line 180: “Set” should be “set”.
- Line 218: “cant” should be “can’t”.

In `OptimisticAsserterInterface.sol`:

- Line 10: “SS” should be “SSM”.

***Update:** Resolved in pull request #4339 with commit 97be5e6.*

Unused import

In `EscalationManagerInterface.sol`, the import `OptimisticAsserterInterface.sol` is unused and could be removed.

***Update:** Resolved in pull request #4338 with commit e4f47ee.*

Inconsistent use of named return variables

Within the `OptimisticAsserter` contract, named return variables are used inconsistently. Most external and public functions in this contract that return a value use a named return variable, but `stampAssertion`, `getMinimumBond`, and `getCurrentTime` do not assign names. None of the internal functions use named variables, making them consistent with each other but inconsistent with the external/public functions. Additionally, the following functions have unused named variables: `assertTruthWithDefaults`, `settleAndGetAssertionResult`, `getAssertion`, and `getAssertionResult`.

When choosing to use named return variables, consider using them in a consistent manner throughout all functions within the same contract.

***Update:** Resolved in pull request #4341 with commit 59364ce. The unused named return variables were removed.*

Use of magic numbers

The `OptimisticAsserter` and `FullPolicyEscalationManager` contracts use the constant value `1e18` in the `settleAssertion` and `getPrice` functions without documenting the value's meaning.

Literal values in the codebase without an explained meaning make the code harder to understand and maintain, thus hindering the experience of developers, auditors, and external contributors alike. To improve the code's readability and facilitate refactoring, consider defining a constant for every magic number, giving it a clear and self-explanatory name.

***Update:** Resolved in pull request #4340 with commit 5ba00f8.*
The `OptimisticAsserter` and `FullPolicyEscalationManager` contracts now independently define a `numericalTrue` constant equal to `1e18` and provide an explanatory comment.

best practices and reduce the potential attack surface. We also recommend implementing monitoring and/or alerting functionality (see Appendix).

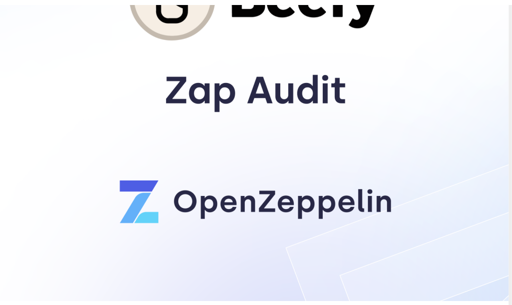
Appendix

Monitoring Recommendations

While audits help in identifying potential security risks, the UMA team is encouraged to also incorporate automated monitoring of on-chain contract activity into their operations. Ongoing monitoring of deployed contracts helps in identifying potential threats and issues affecting the production environment.

- The `_validateAndCacheIdentifier` and `_validateAndCacheCurrency` functions in the `OptimisticAsserter` contract store price identifiers, whitelisted currencies, and associated final fee amounts so they don't need to be retrieved again from the rest of the UMA system. However, these parameters can be invalidated by the `removeSupportedIdentifier`, `removeFromWhitelist`, and `setFinalFee` functions. Consider monitoring the `SupportedIdentifierRemoved`, `RemovedFromWhitelist`, and `NewFinalFee` events to detect potential cache invalidations. A follow-on automated action would be to call `syncUmaParams` and pass in the relevant `identifier` or `currency` value to be updated.
- Consider monitoring the `AssertionMade` event for unusually high `bond` values, which may be indicative of suspicious activity. Because `bond` values much higher than the minimum required bond may be encountered during normal operation of the protocol, this will likely require some data gathering in order to determine where to set the outlier threshold.
- To ensure no unexpected administrative actions are occurring, consider monitoring the `AdminPropertiesSet` event.

Related Posts



Beefy Zap Audit

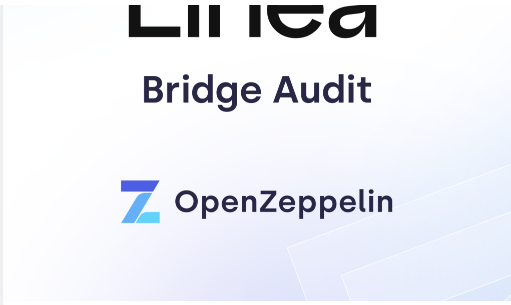
BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

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