UMA Across V2 Audit

MAY 10, 2022 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

Introduction

The UMA Across system provides a mechanism that, in effect, allows users to send funds between all supported chains without waiting for standard token bridge transfers to complete. We audited the UMA Across V2 Protocol over the course of 2 weeks, with 2 auditors, plus another auditor for 1 week.

Scope

The audited commit was `bf03255cbd1db3045cd2fbf1580f24081f46b43a` of the `across-protocol/contracts-v2` repository.

The contracts in scope were (in the `/contracts/` directory):

- `Arbitrum_SpokePool.sol`
- `Ethereum_SpokePool.sol`
- `HubPool.sol`
- `HubPoolInterface.sol`
- `Lockable.sol`
- `LPTokenFactory.sol`
- `MerkleLib.sol`

- `Optimism_SpokePool.sol`
- `Polygon_SpokePool.sol`
- `PolygonTokenBridger.sol`
- `SpokePool.sol`
- `SpokePoolInterface.sol`
- `chain-adapters/Arbitrum_Adapter.sol`
- `chain-adapters/Ethereum_Adapter.sol`
- `chain-adapters/Optimism_Adapter.sol`
- `chain-adapters/Polygon_Adapter.sol`

System Overview

The Across V2 system manages multiple contracts which hold funds and transfer them to each other. These are the `HubPool` and multiple `SpokePools`. The Spokes can exist on other chains, and thus there are standardized "adapters" for sending funds from the hub to the various spokes in order to have a predictable interface.

The system allows users to make deposits on one chain, specifying a desire to withdraw on a different chain and paying a fee. At any point, other users can "fill" this "relay", supplying the original depositor with funds on a different chain and taking a small fee. The relayers are then refunded by the system. If relayers do not fill deposits, the system performs a "slow relay" in which funds are moved across cross-chain bridges to fill the deposit.

The system cannot easily pass messages across cross-chain bridges, so in order for the hub to understand the state of all spokes, and to transfer funds accordingly, merkle trees are produced representing the needed actions, such as rebalances and relayer refunds. These merkle trees are represented with their roots, where the full set of needed merkle roots is called the "root bundle". These are optimistically validated – meaning that they are considered truthful if not disputed within a certain time window. Once the liveness period (in which other users can dispute a root bundle) passes, funds can be transferred between the hub and spokes by using merkle proofs to prove that the transfer was included in the root bundle.

The rules by which a root bundle is determined invalid are notably NOT a part of the smart contract system, and are instead decided by an outside system called the Optimistic Oracle. These dispute rules are to be codified into an UMIP (UMA Improvement Proposal) or multiple UMIPs. Therefore, much of the security of the system rests on the un-audited UMIP, and for the sake of the audit we treated the UMIP as a black box. During the audit, we provided the UMA team with suggestions and reminders for important security considerations when it comes to codifying the UMIP(s).

Privileged Roles

There is one admin for the whole system. This admin can make decisions regarding which chains have valid spokes, which tokens are enabled, and which tokens on some chain map to which tokens on some other chain. The admin also controls parameters such as the system fee percentage, where fees are directed, what the bond for proposing new root bundles is, how disputed root bundles are identified, and which tokens are allowed within the system. This role is intended to eventually be set to the UMA Governor contract (controlled by UMA token holders).

The Optimistic Oracle, which is controlled by UMA holders, has the ability to resolve disputes on root bundles. This means that if it is compromised, it is possible for disputes to not resolve correctly, and, more importantly, whoever can control the optimistic oracle can decide how funds are moved within the system. This is notably a feature of the greater UMA ecosystem, and incentives exist to keep the Optimistic Oracle honest.

Summary

As stated, many of the security properties of the system could not be evaluated as they are affected by UMIPs which are not contained in the scope of this audit. Much of the audit involved checking integrations with cross-chain bridges, and many of the findings in the audit arose from these. Many of the problems identified had to do with problems inherent to synchronising information across multiple chains. More serious issues arose from improper use of signature schemes and insufficient information being passed to distinguish information needed for a single chain when not on that chain.

Overall, we were impressed with the thoughtfulness and attention to edge cases that the UMA team apparently had when developing the protocol. We were also deeply appreciative of their responsiveness when it came to understanding the intent of certain parts of the protocol, and for elucidating the planned UMIP schema for validating root bundles. We appreciated their willingness to collaborate to find solutions and provide documentation to better explain the intent of the codebase.

The UMIP is an extremely crucial part of the system, and if designed poorly creates opportunities for loss of funds in the protocol. The UMIP will need to include robust dispute resolution mechanisms and encompass many different reasons for dispute. Once again, the UMIP was not audited as part of this engagement, though we did provide feedback where applicable to address security concerns that should be addressed by the UMIP.

Finally, there was an issue related to griefing which were identified as an unfortunate byproduct of the system design. The system intentionally does not “earmark” funds for any specific recipient, instead performing rebalances between spokes and allowing authorized users to pull funds from these spokes. Thus, there are potential issues in which a user would have to wait much longer than expected for their funds if the funds are routinely taken by other users before them. However, there is little advantage for an attacker to grief this way, as they pay a small fee to create a valid deposit each time they do. Additionally, this attack goes down in likelihood as liquidity for the specific token increases, as relays for tokens with high liquidity will typically be filled by relayers (instead of system funds) who can earn a profit by doing so. The result is that such greifing is really only a problem for extremely illiquid and centrally held tokens, which may simply not be allowed in the system.

Critical Severity

Slow relays on multiple chains

In each root bundle, the slowRelayRoot represents all the slow relays in a batch, which could involve multiple tokens and spoke pools. A valid root bundle would ensure the `poolRebalanceRoot` has a leaf for every spoke chain. When this rebalance leaf is processed, the `slowRelayRoot` will also be sent to the corresponding spoke pool.

Notably, every spoke pool receives the same `slowRelayRoot`, which represents all slow relays in the batch across the whole system. When the slow relay is executed, the Spoke Pool does not filter on the destination chain id, which means that any slow relay can be executed on any spoke chain where the Spoke Pool has sufficient funds in the `destinationToken`. Consider including the destination chain ID in the slow relay details so the Spoke Pool can filter out relays that are intended for other chains.

Update: *Fixed in pull request #79 as of commit `2a41086f0d61caf0be8c2f3d1cdaf96e4f67f718`.*

Medium Severity

Inconsistent signature checking

Depositors can update the relay fee associated with their transfer by signing a message describing this intention. The message is verified on the origin chain before emitting the event that notifies relayers, and verified again on the destination chain before the new fee can be used to fill the relay. If the depositor used a static ECDSA signature and both chains support the `ecrecover` opcode, both verifications should be identical. However, verification uses the OpenZeppelin Signature Checker library, which also supports EIP-1271 validation for smart contracts. If the smart contract validation behaves differently on the two chains, valid contract signatures may be rejected on the destination chain. A plausible example would be a multisignature wallet on the source chain that is not replicated on the destination chain.

Instead of validating the signature on the destination chain, consider including the `RequestedSpeedUpDeposit` event in the off-chain UMIP specification, so that relayers that comply with the event would be reimbursed. This mitigation would need a mechanism to handle relayers that incorrectly fill relays with excessively large relayer fees, which would prevent the recipient from receiving their full payment. Alternatively, consider removing support for EIP-1271 validation and relying entirely on ECDSA signatures.

Update: *Fixed in pull request #79 as of commit* `2a41086f0d61caf0be8c2f3d1cdaf96e4f67f718`.

Relayers may request invalid repayments

When a relayer fills a relay, they specify a `repaymentChainId` to indicate which chain they want to be refunded on. However, the `repaymentChainId` is not validated against any set of acceptable values. Instead, it is included in the `_emitFillRelay` event, which is used for generating root bundles in the system.

Since not all tokens may exist on all chains, and some chain ID's may not exist or be a part of the Across V2 system, consider specifying valid values for `repaymentChainId` for a given token, and implementing logic similar to that for `enabledDepositRoutes` to use for checking `repaymentChainId`. Alternatively, consider specifying in the UMIP some procedures for root bundle proposers to determine whether a `repaymentChainId` is valid, and what to do if it is not. In this case, invalid `repaymentChainId`s may mean a repayment is simply not repaid – if this is chosen, ensure that this is made very clear in any documentation about the system, so that users are not surprised by losing funds.

Update: *Acknowledged. The UMA team intends to address this off-chain. They state:*

> We believe that this issue can be resolved in a well-defined UMIP that lists valid repayment chain IDs (or points to where to find them), and provide a default repayment chain ID for invalid ones. For example, the UMIP could stipulate that any invalid repayment chain IDs are repaid on mainnet.

Confusing `removeLiquidity` behavior could lock funds

The `removeLiquidity` function in the `HubPool` contract accepts a boolean argument `sendEth`. This should be set to true "if L1 token is WETH and user wants to receive ETH".

However, if the "user" is a smart contract, even if the L1 token is WETH and the `sendEth` argument is `true`, WETH, not ETH, will ultimately be sent back.

This is the case because if `sendEth` is `true`, then the `_unwrapWETHTo` function is called. That function checks if the intended recipient is a smart contract, and, if so, sends WETH.

If the receiving smart contract has no mechanism to handle WETH and was only expecting ETH in return, as was explicitly specified by the `sendEth` argument submitted, then any WETH sent to such a contract could become inaccessible.

To avoid unnecessary confusion and the potential loss of funds, consider either reverting if a smart contract calls `removeLiquidity` with the `sendEth` argument set to `true` or modifying the `_unwrapWETHTo` function so that it can also be provided with and abide by an explicit `sendEth` argument.

Update: *Fixed in pull request #90 as of commit* `a1d1269e8a65e2b08c95c261de3d074abc57444d` *and pull request #139 as of commit* `f4f87583a4af71607bacf7292fee1ffa8fc2c81d`.

`whitelistedRoutes` for `Ethereum_SpokePool` affect other routes

When in `HubPool`'s `executeRootBundle` function, tokens are moved between spokes in order to complete rebalances of the different spoke pools. These token transfers happen within the `_sendTokensToChainAndUpdatePooledTokenTrackers` function, but in order to complete a rebalance the route from the `chainId` of the `HubPool` to the destination chain must be whitelisted.

The issue comes from the conflation of two slightly different requirements. When whitelisting a route, a combination of origin chain, destination chain, and origin token are whitelisted. However, when rebalancing tokens, the specific route where origin chain is the `HubPool`'s chain must be whitelisted for that token and destination chain pairing.

This means that if other routes are to be enabled for rebalancing, the route from the `Ethereum_SpokePool` to some destination chain's `SpokePool` must be enabled as well. This may allow undesired transfers to the `Ethereum_SpokePool`. Additionally, it may cause problems if some token is to be allowed to move between chains aside from Ethereum, but specifically not Ethereum. It would be impossible to disable transfers to the `Ethereum_SpokePool` without also disabling transfers between separate spoke pools for the same token.

Also note that whitelisting a route does not *necessarily* whitelist the route from Ethereum to the same destination chain. This means that a separate transaction may need to be sent to enable rebalances to/from that destination, by whitelisting the Ethereum-as-origin route. This is confusing and could lead to unexpected reversions if forgotten about.

Consider modifying the whitelist scheme so that rebalances to specific chains are automatically enabled when enabling certain routes. For example, if the route for some token to move from Arbitrum to Optimism is enabled, then the route from the Hub to Optimism should also be enabled. Additionally, consider implementing some special logic to differentiate routes from the `HubPool` and routes from the `Ethereum_SpokePool`, so that either route can be enabled independently of the other.

Update: *Fixed in pull request #89 as of commit* `2d0adf78647070e4dd20690f67f46daaa6fc82c4`.

Low Severity

`chainId` function is not `virtual`

Within `SpokePool.sol`, the function `chainId` is marked `override`. However, the comments above it indicate that the function should also be *overridable*, meaning that it should be marked `virtual`.

Consider marking the function `virtual` to allow overriding in contracts that inherit `SpokePool`.

Update: *Fixed in pull request #82 as of commit* `cc48e5721ea444a22a84ddeeef8dcbfe191b112c`.

Lack of input validation

Throughout the codebase there are functions lacking sufficient input validation. For instance:

- In the `HubPool` contract the various admin functions will accept `0` values for inputs. This could result in the loss of funds and unexpected behaviors if null values are unintentionally provided.
- In the `HubPool` contract the `setProtocolFeeCapture` function does not use the `noActiveRequests` modifier. This could allow the protocol fee to be increased even for liquidity providers that have already provided liquidity.
- In the `MerkleLib` library the `isClaimed1D` function does not work as expected if an `index` is greater than 255. In such a case, it will return `true` despite the fact that those values are not actually claimed.
- In the `SpokePool` contract the `deposit` function does not enforce the requirement suggested by the `deploymentTime` comment which says that relays cannot have a quote time before `deploymentTime`.
- In the `SpokePool` contract the `speedUpDeposit` function does not restrict the `newRelayerFeePct` to be less than 50% like the regular deposit does. In practice, the `_fillRelay` function won't accept a fee that is too high, but this should still be enforced within `speedUpDeposit`.
- In the `PolygonTokenBridger` contract the "normal" use case of `send` involves the

caller, `Polygon_SpokePool`, evaluating if the token it is sending is wrapped matic in order to set the `isMatic` flag appropriately. However, for any other caller, if they forget to set this flag while sending wrapped matic, then their tokens would be unwrapped but not sent anywhere. For more predictable behavior, consider checking for wrapped matic inline rather than relying on the `isMatic` argument.

To avoid errors and unexpected system behavior, consider implementing require statements to validate all user-controlled input, even that of admin accounts considering that some clients may default to sending null parameters if none are specified.

Update: *Fixed with pull request #113 as of commit `4c4928866149dcec5bd6008c5ac8050f30898b7f` and pull request #142 as of commit `2b5cbc520415f4a2b16903504a29a9992a63d41c`.*

No good way to disable routes in HubPool

Within the `SpokePool` there exists the `enabledDepositRoutes` mapping, which lists routes that have been approved for deposits (allowing a user to deposit in one spoke pool and withdraw the deposit from another). The `setEnableRoute` function can be used to enable or disable these routes.

Within the `HubPool`, there is a separate `whitelistedRoutes` mapping, which determines whether tokens can be sent to a certain spoke during rebalances. The only way to affect the `whitelistedRoutes` mapping is by calling `whitelistRoute`, which includes a call to enable the `originToken`/`destinationChainId` pair within the Spoke. This means that there is no good way to disable a whitelisted route in the hub without “enabling” the same route in the `enabledDepositRoutes` mapping in the SpokePool.

Assuming that there may be cases in the future where it would be desirable to disable a certain deposit route, consider adding a function which can disable a `whitelistedRoutes` element (by setting the value in the mapping to `address(0)`) without enabling the route in the SpokePool. It may be desirable to disable both atomically from the HubPool, or to establish a procedure to disable them independently in a specific order. Consider designing a procedure for valid cross-chain token transfers in the case that only one mapping has a certain route marked as “disabled”, and including this in the UMIP for dispute resolution. Finally, note that any “atomic” cancellations will still include a delay between when the message is initiated on the hub chain and when execution can be considered finalized on the spoke chain.

Update: *Fixed in pull request #89 as of commit `2d0adf78647070e4dd20690f67f46daaa6fc82c4`.*

Polygon bridger does not enforce `chainId` requirements

The `PolygonTokenBridger` contract’s primary functions are only intended to be called either on l1 *or* l2, but not both. In fact, calling the functions on the wrong chain could result in unexpected behavior and unnecessary confusion.

In the best case, the functions will simply revert if called from the wrong chain because they will attempt to interact with other contracts that do not exist on that chain. For example, calling the `receive` function (by sending the contract some native asset) could trigger reverts on Polygon, but not on Ethereum, because there is a WETH contract at the `l1Weth` address on the latter but not the former.

However, in the worst case, it is possible that such calls will *not* revert, but result in lost funds instead. For example, if a WETH-like contract was later deployed to the `l1Weth` address on Polygon, then the call would not revert. Instead, tokens would be sent to that contract and could remain stuck there.

Although the inline documentation details which function should be called on which chain, consider having the functions in this contract actively enforce these requirements via limiting execution to the correct `block.chainid`.

Update: *Fixed in pull request #115 as of commit `b80d7a5396d31662265bb28b61a1a3d09ed76760` and pull request #128 as of commit `811ac20674d28189fd01297c05ce5b9e89f7a183`.*

Liquidity provisioning can skew fee assessments

In the `HubPool` contract the `enableL1TokenForLiquidityProvision` function allows the contract `owner` to enable an `l1token` to be added to the protocol for liquidity pooling.

This is allowed even if the `l1token` is *already* currently enabled.

As this function also sets the `lastLpFeeUpdate` variable to the then-current `block.timestamp`, enabling an already enabled token will skip over the period of time since `lastLpFeeUpdate` was last set. As a result, any LP fees that should have been assessed for that time period would simply never be assessed.

Consider reverting if this function is called for an `l1token` that is already enabled.

Update: *Fixed in pull request #94 as of commit `b1a097748a82c3276619a06fa36358b574f843e1`.*

Some functions not marked `nonReentrant`

We have not identified any security issues relating to reentrancy. However, out of an abundance of caution, consider marking the following `public` functions in the `HubPool` contract as `nonReentrant`. Consider that the `nonReentrant` modifier only works if both the original function, and the re-entered function are marked `nonReentrant`.

- `setProtocolFeeCapture`
- `setBond`
- `setLiveness`
- `setIdentifier`
- `whitelistRoute`
- `enableL1TokenForLiquidityProvision`
- `disableL1TokenForLiquidityProvision`
- `addLiquidity`

Update: *Fixed. Partially addressed in pull request #62 as of commit `a3b5b5600e53d2ae877a4c1c18d78aadb01ff2e6` and then fully addressed in pull request #92 as of commit `7aa2fa8f46f8d40512857f35dd3ac64587c61f18`.*

Unexpected proposal cancellation

In the `HubPool` contract during a call to the `disputeRootBundle` function, if the `bondAmount` and `finalFee` values are the same, then the proposer bond passed to the optimistic oracle is zero.

When this happens, the optimistic oracle unilaterally sets the bond to the `finalFee` and then attempts to withdraw `bond + final fee`.

Since the `HubPool` only sets the allowance for the oracle to `bondAmount` rather than `bondAmount + finalFee`, this transfer will fail and, as a result, the proposal will be cancelled.

This means that in the situation where `bondAmount` and `finalFee` values are identical, *every* proposal will be cancelled. Consider documenting this situation, checking for it explicitly and reverting with an insightful error message. Additionally, consider trying to avoid the situation by reverting in the `setBond` function if the `newBondAmount` is equal to the `finalFee` or in the `proposeRootBundle` function if `bondAmount` is equal to the `finalFee`.

Update: *Partially fixed in pull request #96 as of commit `671d416db0fe6d813e3761bda0e3132cb30a8e1d`. The condition is checked in `setBond` but not in `proposeRootBundle`.*

Time is cast unsafely

In the `HubPool` function `_updateAccumulatedLpFees`, the return value of `getCurrentTime()` is cast to a `uint32` value. This means that the value will be truncated to fit within 32 bits, and at some point around Feb 6, 2106, it will "roll over" and the value returned by casting to `uint32` will drop down to `0`. This will set `pooledToken.lastLpFeeUpdate` to a much lower number than the previous `lastLpFeeUpdate`. Any subsequent time `_getAccumulatedFees` is called, the `timeFromLastInteraction` calculation will be exceedingly high, and all "undistributed" fees will be accounted for as accumulated.

Again, note that this issue will only occur starting in the year 2106. Consider changing the size of the cast from `uint32` to a larger number, like `uint64`. This should be more than enough to not encounter limits within a reasonably distant future. Alternatively, consider documenting the behavior and defining a procedure for what to do if the system is still in operation when the `uint32` limit is hit, or for shutting down the system before the year 2106.

Update: *Fixed in pull request #95 as of commit `2f59388906346780e729f2b879b643941ea314c9`.*

Notes & Additional Information

Missing link to referenced code

Within the `Ethereum_Adapter`, there is a mention of copying code from "Governor.sol". It appears that the contract in question is `Governor.sol` from the `UMAprotocol/protocol` repository.

Since it is a part of a separate repository, and it is possible that the code may change in the future, consider including a link to the file, including a commit hash, so that it can be easily referenced by developers and reviewers in the future.

Update: *Fixed in pull request #97 as of commit `ac9ed389914dc4249f488226fcd94d6d0b44aeb0`.*

Inconsistent approach to `struct` definitions

The `PoolRebalanceLeaf` struct is defined in `HubPoolInterface.sol`, while the `RootBundle`, `PooledToken`, and `CrossChainContract` structs are all defined in the implementation, `HubPool.sol`.

Consider defining all `struct`s for `HubPool` within the same contract.

Update: *Fixed in pull request #100 as of commit `9a98ce1ae5c8c5e95bcfa979666b980008d14d3f`.*

Inconsistent token metadata versioning

In the `LpTokenFactory` contract, the LP tokens it creates have inconsistent versioning in their metadata.

While the token `symbol` is prepended with `Av2` (ostensibly for "Across version 2"), the token `name` is prepended only with "Across" and no version number.

Consider adding the version number to the token `name`, or, alteratively, leaving an inline comment explaining the decision to omit the version number.

Update: *Fixed in pull request #101 as of commit `91a08a9bd2b47a1a1319aff8bda53349e8264ce3`.*

Lack of documentation

Although most of the codebase is thoroughly documented, there are a few instances where documentation is lacking. For instance:

- In the `HubPool` contract the public `unclaimedAccumulatedProtocolFees` variable has no inline documentation.
- In the `HubPoolInterface` contract the inline documentation accompanying `PoolRebalanceLeaf.netSendAmounts`, although lengthy, could benefit from additional clarification around the case of negative values. It could clarify further that in such cases the actual `netSendAmounts` value is ignored, but it should match the `amountToReturn` parameter in the `RelayerRefundLeaf`.
- Many of the functions in the `MerkleLib` library are missing NatSpec `@return` statements.

To further clarify intent and improve overall code readability, consider adding additional inline documentation where indicated above.

Update: *Fixed in pull request #102 as of commit `e2bfe128ff1a9aeed02bfcebe58a5880ad283698`.*

Magic values

In the `LpTokenFactory` contract, when the `createLpToken` function is called, it creates a new ERC20 LP token and adds the `msg.sender` to the new token's `minter` and `burner` roles. These role assignments use the magic values `1` and `2`, which are the `uint` identifiers for the respective roles.

Rather than using these literal values to assign roles, consider using the the `ExpandedERC20.addMinter` and `ExpandedERC20.addBurner` functions.

Update: *Fixed in pull request #103 as of commit `e9d3419ac6eb609b0c9165cdeac3fbff58285d18`.*

Misleading Comments

- `HubPool` lines 718-719 explain that the `whitelistedRoute` function returns whitelisted destination tokens, but does not mention that if the token is *not* whitelisted then the function returns `address(0)`.
- The comments in the declaration of the `PoolRebalanceLeaf` struct appear to refer to a previous version of the struct, making them hard to follow. For example, line 17 implies there are two arrays above it (there is only one), and line 31 suggests there are multiple arrays below it (there is only one).
- A comment about `HubPool.executeRootBundle` states that the function deletes the published root bundle, however it does not.
- Within the `LPTokenFactory` contract, the comments on lines 24 and 25 should say "msg.sender" or "the calling contract" rather than "this contract".
- The comments above the `lpFeeRatePerSecond` variable suggest that LP fees are released linearly. In fact, they are released sublinearly, because the `_getAccumulatedFees` function uses a fraction of the `undistributedLpFees` (which decreases over time for any given loan), rather than the total funds on loan.
- The comment in `SpokePool` above the definition of `claimedBitmap` state that there are `256x256 leaves per root`. However, due to the indexing scheme in `MerkleLib`, there are a maximum of `2^248` different values of `claimedWordIndex`, with `256` different `claimedBitIndexes`. A more clear comment might explain that there are `256x(2^248)` leaves per root.

Consider correcting these comments to make the code easier to understand for reviewers and future developers.

Update: *Fixed in pull request #109 as of commit `21cdccd5cbfffd4f120ab56c2691b8e961a8d323`, pull request #104 as of commit `1148796377365a2de52fb89810f769ffb7f8c96f` and pull request #138 as of commit `c0b6d4841b86ba8acf3e4a3042a78a1307410e6a`.*

`payable` `multicall` function disallows `msg.value`

The `MultiCaller` contract is inherited by the `HubPool` and `SpokePool` contracts. It provides the public `multiCall` function that facilitates calling multiple methods within the same contract with only a single call.

However, although it is designated as a `payable` function, it disallows any calls that send ETH, ie where `msg.value` is not zero.

This effectively makes the `payable` designation moot and the contradictory indications could lead to confusion.

In the context of the `HubPool`, specifically, relays destined for chains where ETH is required and where a call to `loadEthForL2Calls` is therefore necessary, will not be multi-callable.

Consider either explicitly noting this limitation, or removing both the `require` statement *and* the `payable` designation.

Update: *Fixed in pull request #98 as of commit* `7092b8af1da15306994ea760b9669a9bd1f776c1`.

Naming issues

We have identified some areas of the code which could benefit from better naming:

- In `HubPoolInterface.liquidityUtilizationPostRelay`, the parameter `token` should be renamed to `l1Token` to better match other functions in the interface, as well as the function's implementation in `HubPool`.
- In the `RootBundle` struct, `requestExpirationTimestamp` should be renamed to better indicate that it ends the "challenge period". Consider renaming it to `ChallengePeriodEndTimestamp` or similar.
- The `RootBundleExecuted` event in `HubPool.sol` only names `one of its array parameters` in the plural form, but when the event is emitted, all array parameters are named in the plural form. Consider changing the event definition so that all array parameters are pluralized.
- The name of `function whitelistedRoute` is vague and does not indicate what it's output will be. Consider renaming it to something like `destinationTokenFromRoute` to better match the return value.
- When `weth` is used in `Polygon_SpokePool.sol`, it refers to wrapped MATIC. Consider renaming the `weth` variable in `SpokePool.sol` to `wrapped_native_token` to make it more generalizable. This will make `Polygon_SpokePool` less confusing and be more generalizeable for future SpokePools.
- The `executeSlowRelayRoot` and `executeRelayerRefundRoot` functions are executing leaves and should be renamed accordingly.
- The `unclaimedPoolRebalanceLeafCount` parameter of the `ProposeRootBundle` event should be renamed to `poolRebalanceLeafCount`, since it's always the total number of leaves in the tree.
- The `RootBundleCanceled` event names the last parameter as `disputedAncillaryData`, but the proposal is not necessarily disputed. It should just be `ancillaryData`.
- The `_append` function of the `LpTokenFactory` could be called `_concatenate` to better describe its functionality.
- The `onlyEnabledRoute` modifier has a `destinationId` parameter that should be `destinationChainId` to match the rest of the code base.

Consider following our renaming suggestions to make the codebase easier for developers and reviewers to understand.

Update: *Fixed in pull request #105 as of commit* `87b69cdf159a1db5ccfcaa9f27825dfa416e7158`.

Warning about nonstandard tokens

Although tokens must be enabled to be used in the system, it is important to define what may make a token troublesome so that which tokens can be whitelisted is easier to determine.

- ERC20 tokens which charge fees, or which can charge fees, will result in various accounting issues as the amount `transferred` will not match the amount received by the contracts in the system. Many spots in the code, such as in the `addLiquidity` function, assume the amount transferred in equals the amount received.

- ERC777 tokens, which are ERC20-compatible, include hooks on transfers. These hooks are configurable and may be configured to revert in some or all cases. In `SpokePool._executeRelayerRefundRoot`, a failing transfer for one token could block all other refunds for the specified leaf.
- Tokens which are upgradeable may change their implementations to become subject to the above issues, even though they may not have been problematic before being upgraded.

Consider documenting procedures for tokens which behave unexpectedly to be filtered for before whitelisting.

Update: *Fixed in pull request #137 as of commit `ba6e03974cf722d33b9fb2def4da578129f5baed`.*

Not using `immutable`

Within the `HubPool` contract, the `weth`, `finder`, and `lpTokenFactory` variables are only ever assigned a value in the constructor.

Consider marking these values as `immutable` to better signal the fact that these values or not meant to change and to reduce the overall gas consumption of the contract.

Update: *Fixed in pull request #108 as of commit `cccb9556345edcc5d8fc3022ab64a5b368c8d810`.*

Residual privileged roles

When the `LpTokenFactory` contract creates an `ExpandedERC20` token contract, the factory becomes the `owner` of that token contract. The factory then proceeds to assign the `minter` and `burner` roles to the `msg.sender`. The factory remains the `owner`.

As this is a residual power that is no longer needed by the `LpTokenFactory`, consider reducing the number of addresses with privileged roles by transferring ownership to the `msg.sender`.

Update: *Fixed in pull request #109 as of commit `21cdccd5cbfffd4f120ab56c2691b8e961a8d323`.*

Typographical errors

In `HubPool.sol`:

- line 99: “Heler” should be “Helper”
- line 201: “proposal” should be “Proposal”
- line 235: “its” should be “it’s”
- line 294: “disputes..” should be “disputes.”
- line 377: “for again” should be “again.”
- line 419: “access more funds that” should be “to access more funds than”
- line 475: “to along” should be “along”
- line 480: “leafs” should be “leaves”
- line 532: “neccessary” should be “necessary”
- line 568: “to back” should be “back”
- line 569: “leafs” should be “leaves”
- line 569: “wont” should be “won’t”

- line 865: “timeFromLastInteraction ,undistributedLpFees)” should be “timeFromLastInteraction, undistributedLpFees)”
- line 866: “a fees.” should be “fees.”
- line 913: “decrease” should be “decreased”
- line 962: “send” should be “sent”

In `HubPoolInterface.sol`:

- line 13: “sent” should be “send”

In `MerkleLib.sol`:

- line 86: “*” should be “*”

In `Polygon_SpokePool.sol`:

- line 43: “priviledges” should be “privileges”

In `SpokePool.sol`:

- line 55: “token” should be “chain”
- line 67: “leafs” should be “leaves”
- line 292: “users” should be “user’s”
- line 347: “receipient.” should be “recipient.”

In `SpokePoolInterface.sol`:

- line 11: “inverted.” should be “negated.”
- line 27: “a the” should be “the”

Update: *Fixed in pull request #110 as of commit* `813cfeef126484e0ac5b7fb91225560c5edbff7c`.

Undocumented implicit approval requirements

Throughout the codebase, when the `safeTransferFrom` function is used to transfer assets into the system from an external address there is an implicit requirement that the external address has already granted the appropriate approvals.

For instance:

- The `proposeRootBundle` function relies on `safeTransferFrom` which requires that `HubPool` has been granted an allowance of `bondAmount` `bondToken`s by the caller.
- The `addLiquidity` function relies on `safeTransferFrom`, requiring that the `HubPool` has been granted an `l1TokenAmount` allowance of the caller’s `l1Token`.

In favor of explicitness and to improve the overall clarity of the codebase, consider documenting all approval requirements in the relevant functions’ inline documentation.

Update: *Fixed in pull request #111 as of commit* `5a3ef77a22b81411a3616bb48acf063acabb4d2c`.

Unused code

Throughout the codebase, there are instances of unused code. For example:

- The `proposerBondRepaid` attribute of the `HubPool` contract's `RootBundle` struct is never used. Consider removing it.
- The events in the `Arbitrum_Adapter` contract are never used. As the relevant state variables are `immutable`, consider setting all relevant values in the constructor and emitting these events then. Alternatively, consider adding comments indicating why events are declared but unused.
- The `L2GasLimitSet` event in the `Optimism_Adapter` is never emitted. Consider emitting it in the constructor, removing it, or adding a comment indicating why it is declared but not used.
- The `HubPoolChanged` event is never used.

Update: *Fixed in pull request #78 as of commit* `f7e8518050a12e478516da6622bcf2357bb2e802` *and in pull request #99 as of commit* `d89b1fb8d491703ef63dae0b29d93abd29d501de`.

Unnecessary import statements

The below list outlines contract import statements that are unnecessary:

- The `WETH9` and `Lockable` imports are not used in the `Ethereum_Adapter` contract.
- The `CrossDomainEnabled`, `IL1StandardBridge`, and `Lockable` imports are not used in the `Polygon_Adapter` contract.
- The `WETH9` and `IERC20` imports are not used in the `Arbitrum_Adapter` contract.
- The `AdapterInterface` interface is imported twice in the `Arbitrum_Adapter` contract.
- The `WETH9` and `SpokePoolInterface` imports are not used in the `Ethereum_SpokePool` contract.
- The `IERC20` import in the `LpTokenFactoryInterface` interface is unused.
- The `MerkleLib` is imported twice in the `SpokePool` contract.

Consider removing unnecessary import statements to simplify the codebase and increase overall readability.

Update: *Fixed in pull request #112 as of commit* `d81295d3fd433a1f08fdd42c75a0aa3233a77dbe`.

`whitelistedRoute` can be `external`

The `whitelistedRoute` function within `HubPool` is marked as `public`. However, it is not called anywhere within the codebase.

Consider restricting the function to `external` to reduce the surface for error and better reflect its intent.

Update: *Fixed in pull request #89 as of commit* `2d0adf78647070e4dd20690f67f46daaa6fc82c4`.

Conclusions

One critical issue was found. Some changes were proposed to follow best practices and reduce the potential attack surface. The contracts are highly dependent on a well-structured UMIP which determines the behavior of the Optimistic Oracle.

Update: Additional PRs reviewed

During the fix review process, the UMA team provided us with a list of additional pull requests for our review. We proceeded to review the following additional PRs related to the Across V2 codebase:

- Pull request #78 as of commit `f7e8518050a12e478516da6622bcf2357bb2e802` added "Emergency admin features to pause proposals and executions of root bundles, and to delete root bundles from the spoke pool to prevent a single bad bundle from permanently breaking or disabling the system."

- A single security concern was noted: the Check-Effects-Interactions pattern was not being employed for the newly introduced `emergencyDeleteProposal` function. We raised that this is counter to best practice and could potentially, lead to issues later. This was then addressed later in pull request #147 as of commit `ee7714734aab4ed0457c813403a63e53c6438529`.
- Pull request #77 as of commit `8cf240a147b7d0467418eb81b2d6e152d478d101` removes an extraneous fee. Specifically, it addresses the fact that the: “Slow relay charges 0 relayer fee % and refunds user this fee. The relayer fee is intended as a speed fee. The user shouldnâ€™t pay this fee for capital that the relayer doesnâ€™t loan them.”
- No security concerns were noted.
- Pull request #76 as of commit `70c56813e908cb5d02c43501d7de6a2c01564dca` made changes to prevent a Spoke Pool’s address from accidentally/intentionally being set as `address(0)`.
- No security concerns were noted.
- Pull request #64 as of commit `029406ec534da9979b63acf354e63394b4ce3a90` changed the sizes of various `uint`s to better limit their range of values and to prevent them from holding values which are too high. This is related to issue **L02**.
- No security concerns were noted.
- Pull request #65 as of commit `d2ca5b2f1f604e30083a20c72f40d971c4161c59` added a mapping to allow tokens on Optimism to be transferred across custom bridges rather than the standard bridge.
- No security concerns were noted.
- One suggestion to allow the blank `data` field to be populated was made, but ultimately decided against.
- Pull request #85 as of commit `248bb4d67dfb195b7077f8632f548fa3db808be5` added logic to prevent redundant relays of root bundles to spoke pools on L2.
- No security concerns were noted.
- Pull request #120 as of commit `a09e56b554577da8b929d8043fc6cdfb654e2ecf` made changes to fix reversions when transferring tokens to Arbitrum.
- No security concerns were noted.
- Pull request #128 as of commit `811ac20674d28189fd01297c05ce5b9e89f7a183` made changes to fix token bridging transactions using Polygon’s two bridge implementations.
- No security concerns were noted.
- Pull request #67 as of commit `ac18f6a3fc89bc861af183a0b731c89837cf84ba` modified parameter indexing for events.
- No security concerns were noted.
- Pull request #81 as of commit `a72519e0965fc298ada2d19942ec5806530988df` implemented argument spreading rather than passing `PoolRebalanceLeaf` objects when executing a root bundle “to improve etherscan tx processing.”
- No security concerns were noted.
- Pull request #84 as of commit `3ec3a7f990ee9a50a4a44f6baf893d38d2914b38` removed `getRootBundleProposalAncillaryData` functionality based on the fact that, even with the prior implementation of the function, off-chain information will still be required to dispute proposals.
- No security concerns were noted.
- Noted that `AncillaryData.sol` is still being imported in `HubPool.sol`, though no longer used.
- Pull request #114 as of commit `5a31be8aac645085f59e20cbb17e2fb24ec24f85` removes the `getRootBundleProposalAncillaryData` function altogether since it was just returning a literal empty string.
- No security concerns were noted.
- Pull request #116 as of commit `30ea0888b141c4085d7e30eab4beecd6c8fd9a62` bumped the Solidity compiler version to the latest.

- No security concerns were noted.
- Pull request #73 as of commit `98237643f482d9333b394cbf3f2a2c075205b7ba` made changes related to gas optimizations and storage packing.
- No security concerns were noted.
- Noted unnecessary `uint32` usages in for loops that increased gas consumption and unnecessarily increased the possibility for overflow. This concern was subsequently addressed in pull request #148 as of commit `f6d5bc387d24da6fc1cd99de10700d744daf3f6a`.
- Pull request #119 as of commit `709bf1d99e32e5a3bea7605c218020e9d6a1e1f5` suppressed solhint warnings (in as limited a manner as possible).
- No security concerns were noted.
- Noted a lack of spacing in some of the solhint suppression directives.

RELATED POSTS