OpenZeppelin | security

Taiko Protocol Audit

taiko

June 19, 2024

Table of Contents

Summary

Type	Rollup
Timeline	**(Phase 1)** From 2024-03-04 To 2024-04-05 **(Phase 2)** From 2024-04-08 To 2024-05-03 **(Phase 3)** From 2024-05-13 To 2024-05-28
Languages	Solidity

Total Issues	62 (47 resolved, 8 partially resolved)
Critical Severity Issues	2 (2 resolved)
High Severity Issues	3 (2 resolved, 1 partially resolved)
Medium Severity Issues	9 (8 resolved)
Low Severity Issues	23 (19 resolved, 2 partially resolved)
Notes & Additional Information	23 (14 resolved, 5 partially resolved)

Scope

The audit took place in three phases.

Phases 1 and 2

During the first two phases, we audited the taikoxyz/taiko-mono repository at commit b47fc34.

In scope were the following files:

```
└── packages
    └── protocol
        └── contracts
            ├── bridge
            │   ├── Bridge.sol
            │   └── IBridge.sol
            ├── common
            │   ├── AddressManager.sol
            │   ├── AddressResolver.sol
            │   ├── EssentialContract.sol
            │   ├── IAddressManager.sol
            │   └── IAddressResolver.sol
            ├── L1
            │   ├── ITaikoL1.sol
            │   ├── TaikoData.sol
            │   ├── TaikoErrors.sol
            │   ├── TaikoEvents.sol
            │   ├── TaikoL1.sol
            │   ├── TaikoToken.sol
            │   ├── gov
            │   │   ├── TaikoGovernor.sol
            │   │   └── TaikoTimelockController.sol
            │   ├── hooks
            │   │   ├── AssignmentHook.sol
            │   │   └── IHook.sol
            │   ├── libs
            │   │   ├── LibDepositing.sol
            │   │   ├── LibProposing.sol
            │   │   ├── LibProving.sol
            │   │   ├── LibUtils.sol
            │   │   └── LibVerifying.sol
            │   ├── provers
            │   │   ├── GuardianProver.sol
            │   │   └── Guardians.sol
            │   ├── tiers
            │   │   ├── DevnetTierProvider.sol
            │   │   ├── ITierProvider.sol
            │   │   ├── MainnetTierProvider.sol
            │   │   └── TestnetTierProvider.sol
            ├── L2
            │   ├── CrossChainOwned.sol
            │   ├── Lib1559Math.sol
            │   ├── TaikoL2.sol
            │   └── TaikoL2EIP1559Configurable.sol
            ├── libs
            │   ├── Lib4844.sol
            │   ├── LibAddress.sol
            │   ├── LibMath.sol
            │   └── LibTrieProof.sol
            ├── signal
            │   ├── ISignalService.sol
            │   ├── LibSignals.sol
            │   └── SignalService.sol
            ├── team
            │   ├── TimelockTokenPool.sol
            │   └── airdrop
            │       ├── ERC20Airdrop.sol
            │       └── MerkleClaimable.sol
            ├── thirdparty
            │   ├── nomad-xyz
            │   │   └── ExcessivelySafeCall.sol
            │   ├── optimism
            │   │   ├── Bytes.sol
            │   │   ├── rlp
            │   │   │   ├── RLPReader.sol
            │   │   │   └── RLPWriter.sol
            │   │   └── trie
            │   │       ├── MerkleTrie.sol
            │   │       └── SecureMerkleTrie.sol
            │   └── solmate
            │       └── LibFixedPointMath.sol
            ├── tokenvault
            │   ├── BaseNFTVault.sol
            │   ├── BaseVault.sol
            │   ├── BridgedERC1155.sol
            │   ├── BridgedERC20.sol
            │   ├── BridgedERC20Base.sol
            │   ├── BridgedERC721.sol
            │   ├── ERC1155Vault.sol
            │   ├── ERC20Vault.sol
            │   ├── ERC721Vault.sol
            │   ├── IBridgedERC20.sol
            │   ├── LibBridgedToken.sol
            │   └── adapters
            │       └── USDCAdapter.sol
            └── verifiers
                ├── GuardianVerifier.sol
                ├── IVerifier.sol
                └── SgxVerifier.sol
```

Phase 3

During the third phase, we audited the taikoxyz/taiko-mono repository at commit dd8725f.

In scope were the following files:

```
└── packages
    └── protocol
        └── contracts
            ├── L2
            │   └── DelegateOwner.sol
            ├── bridge
            │   ├── Bridge.sol
            │   └── QuotaManager.sol
            ├── tko
            │   └── BridgedTaikoToken.sol
            └── tokenvault
                ├── BridgedERC20.sol
                └── ERC20Vault.sol
```

System Overview

Taiko is a rollup that posts its data to Ethereum and relies on Ethereum smart contracts to store its state and validate state transitions. The validity of these state transitions is enforced through a tier and contestation system, where higher proof tiers can be used to contest a transition with a lower tier. In addition, sequencing transactions and building L2 blocks is permissionless. This means that L1 validators effectively choose which blocks are proposed as well as the order of L2 transactions, making Taiko a based rollup.

We will describe the system below assuming only Ethereum L1 and a single Taiko L2 for simplicity, but we note that the codebase was built to be extendable to additional rollups, including on higher layers.

`TaikoL1`

The `TaikoL1` contract is the core of the Taiko protocol, providing functionalities for proposing, proving, and verifying L2 blocks. As noted above, it was designed to operate on L1 but can be adapted for deployment on other layers, enabling the creation of additional layers like L3s. Central to its operations, `TaikoL1` manages the entire life cycle of L2 blocks. The cycle includes:

Block Proposal

Block proposers select transactions from the Taiko L2 mempool to form a block, incentivized by L2 transaction tips and potential MEV. The proposer then submits these transactions in blobs or calldata to the `TaikoL1` contract. This submission also includes additional details such as the assigned prover, a list of addresses (called "hooks") called during submission and some additional data sent to the hooks. The `AssignmentHook` is one such hook which is deployed by the Taiko team, allowing third-party provers to express their consent through a signature and receiving some fees paid by the proposer as compensation. Provers are tasked with computing the proofs required to prove transitions associated with blocks, which is expected to require significant computing power and custom infrastructure. In exchange for their monopoly over proving a block for a limited amount of time, assigned provers have to provide some Taiko (TKO) tokens which are held by the `TaikoL1` contract, called a "liveness bond". This bond is returned if the prover proves the correct transition in the imparted time.

Block Proving

As mentioned above, the assigned prover must send a proof to validate the block transition within a designated time window. If the prover fails to do so, their liveness bond will be given as an incentive for another provers to prove the block. If the prover submits the proof on time, it enters a cooldown period during which it can be contested by others. Contestants are not required to provide any proof but must pay a "contestation bond" in TKO tokens. If a proof is contested, a higher-tier proof is required to resolve the dispute before the associated block can be verified.

Block Verification

After the cooldown period has passed, a block can no longer be contested and is assumed to be valid. While a new block is being proposed, previous ones are verified once their cooldown period is over. After this process, the L2 state is finalized and can be used to facilitate the bridging process between L1 and L2.

Overview of Multi-Proofs System

As cryptographic methods are continually evolving and carry inherent risks, Taiko has adopted a multi-tiered approach to proofs. The first tier is SGX proofs, the second tier requires both SGX and ZK proofs, and the last tier requires proofs provided by a whitelisted set of guardians. This tiered system, combined with economic incentives, aims to make all of Taiko's state transitions as sound as zk proofs. This is because zk provers are incentivized to contest any invalid SGX proof. At the same time, as SGX proofs are cheaper to compute than zk proofs, block submission and transaction fees paid by users are expected to be cheaper on average than pure zk rollups. This tier system mitigates risks associated with potential vulnerabilities in proof systems and allows for configurational flexibility that can evolve into a standard zk-rollup over time.

Prover Economics

To sustain a pool of zk provers, Taiko has implemented a system whereby a minimum proof tier is randomly assigned to each new block, thus incentivizing zk provers to monitor and prove L2 blocks. In addition, provers are incentivized through bonds to contest invalid proofs.

Guardian Provers

As the multi-proofs system gets battle-tested during the first phase of the protocol's life, the guardian provers act as a fallback against unforeseen bugs within the proving systems. They have the power to prove arbitrary state transitions in any block. These guardian provers operate externally to the main protocol and can be removed from the multi-tier proving setup. As the system evolves and other provers demonstrate reliability, the necessity for these guardian provers should diminish over time.

`TaikoL2`

The `TaikoL2` contract is one of the main components of the Taiko rollup and is part of the contracts that are going to be deployed on the L2 side. This contract has two core functionalities.

Firstly, it facilitates cross-layer communication by "anchoring" the L1 state root onto L2. This functionality enables users on L2 to demonstrate that a specific message was sent on L1 by leveraging the anchored L1 state root along with a Merkle proof.

Secondly, it incorporates a slightly modified version of the exponential EIP-1559 mechanism to manage L2 gas pricing. EIP-1559 is known for its dynamic fee computation which adjusts the base fees based on network congestion, thus stabilizing congestion and transaction fees. In `TaikoL2`, some parameters of this base fee calculation are configurable by the Taiko DAO. A fixed amount of L2 gas is made available per L1 block.

During phases 1 and 2, the `TaikoL2` contract inherited from the `CrossChainOwned` contract to enable certain functions to be executed from L1 through a cross-chain interaction. This was simplified in phase 3 by making the `DelegateOwner` the owner of all the L2 contracts. This `DelegateOwner` contract is operated by an owner on L1 through cross-chain messages, allowing it to call other contracts in the name of the `DelegateOwner`. In practice, this allows the L1 DAO to effectively act as the owner of the L2 contracts.

Cross-Chain Communication

At the core of cross-chain communication is the `SignalService` contract. This contract stores as parts of its state the signals that are used to communicate across chains. There are two main types of signals: signals sent by contracts to communicate with a different chain, such as the `Bridge` contract, and signals sent by the `SignalService` contract itself to indicate that the state/storage root for another chain has been synchronized with this chain. These synchronization operations happen, for example, during L2 block verification on L1, or in the anchor transaction at the beginning of new blocks on L2.

Once a state root for a chain has been synchronized on another chain, a Merkle proof can be done against this state root to prove that a certain signal was sent on the source chain. Such proofs can also use multiple "hops": for example, a user could prove that a signal was sent on L3 if the state of the L3 was synchronized to L2 and the L2 state was synchronized to L1. This could be used to execute an L3 -> L1 message in one L1 transaction using two hops. This proving mechanism underlies Taiko cross-chain communication.

The `SignalService` can be used by other contracts such as the `Bridge` contract to handle cross-chain communication. Messages can be sent to the `Bridge` contract on the source chain, and then processed on the target chain once the state has been synchronized. A fee can be associated with a message to incentivize third-party actors to process it, or some parameters can be set on the message so that it can only be processed by a specific address. Messages can be used to call contracts and can be retried if they fail. When retrying a failed message, its status can be updated so that it can no longer be retried and the sender on the source chain can receive a refund.

Vaults

There are three types of vaults: The `ERC20Vault`, the `ERC721Vault`, and the `ERC1155Vault`. As their names suggest, each of these is responsible for handling a certain type of asset. These vaults use the `Bridge` contract mentioned above to send messages to a vault on another chain and hold all the sent assets. When a token is sent for the first time across the chain, a new bridged contract (e.g., `BridgedERC20`) is deployed on the target chain to represent it. A migration can be triggered by the owner of a vault to change the contract associated with a certain bridged asset. When a migration is triggered, the previous asset is blocked from being transferred through the vault and users have to migrate to the new asset. As part of phases 1 and 2, a special contract called `USDCAdapter` was designed to handle the migration from the initial USDC `BridgedERC20` representation to a native USDC contract on Taiko. This adapter was removed in phase 3 as the `mint` and `burn` functions of `IBridgedERC20` interface were changed to match USDC's. Other custom native tokens being deployed to L2 may still require adapters.

Quota Manager

The `QuotaManager` contract was added to the codebase as part of phase 3 following the auditors' advice during previous phases. The contract allows the owner to define quotas (spending limits) which are intended to be set up for ERC-20 tokens and ETH. All quotas are recovering over the same quota period. The `Bridge` and `ERC20Vault` consume these quotas when forwarding assets, with the `QuotaManager` tracking the available quota and preventing limits from being exceeded. These rate limits are expected to be relaxed as the protocol matures.

Governance

During phase 1 and 2, the contracts in the codebase were intended to be owned by the `TaikoTimelockController` contract. This contract sat between the `TaikoGovernor` contract, which inherits from OpenZeppelin's `Governor`, and the owned contracts. The `TaikoGovernor` contract was intended to be used by the DAO composed of `TaikoToken` holders to vote on proposals, which would then be executed on-chain through the `TaikoTimelockController` contract, adding a delay for users in case of a malicious action by governance. The DAO and the timelock delay could be circumvented by an address set during initialization of the `TaikoTimelockController` contract (e.g., a security council).

As part of phase 3, these contracts were removed and L1 contracts are now owned directly by the DAO. L2 contracts are owned by the `DelegateOwner` contract which is operated by the L1 DAO by sending L1->L2 messages.

Management of Special Roles

The `AddressManager` and `AddressResolver` contracts are central to the operations of the Taiko ecosystem. The `AddressManager` contract allows the contract's owner, in practice the Taiko DAO, to define and update the address associated with a unique chain ID and name pair through the `setAddress` method. This configuration allows any contract inheriting from the `EssentialContract` — which itself inherits from the `AddressResolver` — to perform name-to-address lookups. Such functionality is important for ensuring that contracts can reliably utilize the same designated addresses within their operational logic.

The `AddressResolver` contract enables straightforward address retrieval from the `AddressManager` contract. This separation of concerns simplifies address resolution and enhances the system's flexibility in managing addresses, ensuring that any address change affects all the contracts simultaneously by maintaining a unique source of truth per layer.

`EssentialContract`

`EssentialContract` serves as a foundational component in the Taiko system, with all the Taiko contracts inheriting from it. This abstract contract incorporates several key features:

- **Owner Management**: Utilizing the `Ownable2StepUpgradeable` contract, `EssentialContract` provides a basic access control mechanism, where an account (an owner) can be granted exclusive access to specific functions. Importantly, ownership can be transferred through a two-step transaction process, enhancing security by reducing risks associated with single-transaction ownership transfers.
- **Upgradability**: Through inheritance from `UUPSUpgradeable`, the contract supports the UUPS proxy pattern.
- **Pausing Mechanism**: The contract grants the owner the power to pause and unpause the execution of the functions that use pausing modifiers.
- **Reentrancy Protection**: The inclusion of a `nonReentrant` modifier prevents reentrancy attacks.
- **`AddressManager` Access**: `EssentialContract` optionally enables the `AddressResolver`, allowing child contracts to query and use resolved addresses dynamically within their functional logic. This capability is enhanced with the `onlyFromOwnerOrNamed` modifier, restricting the execution of certain functions to either the owner or specific named addresses within the `AddressManager`.

Taiko Grants

The `TimelockTokenPool` was a contract within the Taiko ecosystem that was audited in phases 1 and 2. It was designed for managing the distribution and lifecycle of TKO tokens to various stakeholders such as investors, team members, advisors, and grant program grantees.

The contract delineates a three-stage lifecycle for token management:

1. **Allocated**: Tokens are granted for specific roles or individuals but remain under the control of the shared vault.
2. **Granted, Owned, and Locked**: At this stage, tokens are officially owned by the recipients but are locked according to predefined conditions.
3. **Granted, Owned, and Unlocked**: In the final phase, the tokens transition from being locked to fully unlocked based on the terms set out at the time of the grant. Once unlocked, the recipients can withdraw all their assets.

The contract includes functionality to void conditional allocations by invoking the `void()` method. However, once tokens are granted and owned, the terms of the ownership and unlock schedules are immutable, ensuring clarity and fairness in the distribution process. To accommodate different groups of recipients, the contract can be deployed in multiple instances.

Privileged Roles

Multiple privileged roles can interact with the system:

- Guardians: Guardians are managed by the `GuardianProver` contract. They monitor block proposals and can execute arbitrary L2 state transitions for proposed blocks.
- Owner: All the contracts have an owner who can pause functionality in some contracts, call privileged functions, and upgrade the implementation.
- `proposer`: If the `proposer` role is given to any contract through the address manager, this address is the only one that can propose a block. The owner of the `AddressManager` contract can grant this role.
- `proposer_one`: Similar to the `proposer` role above, an address with this role is the only address which can propose the first block. This role is also granted by the owner of the `AddressManager` contract.
- `TIMELOCK_ADMIN_ROLE`: Any address given this role in the `TaikoTimelockController` contract can execute transactions instantly and circumvent the timelock delay. Note that the `TaikoTimelockController` contract was expected to be given ownership role of the other contracts, giving the `TIMELOCK_ADMIN_ROLE` effective ownership of the other contracts. As mentioned above, this role and the `TaikoTimelockController` contract were removed in phase 3.
- `bridge_watchdog`: In phases 1 and 2, addresses with this role could prevent the processing of some messages on the `Bridge` contracts, as well as prevent some addresses from being called by messages. In phase 3, this role can pause the `Bridge` contract but cannot unpause it.
- `bridge_pauser`: Addresses with this role can pause the bridge, preventing users from sending or processing messages.
- `chain_pauser`/`chain_watchdog`: In phases 1 and 2, addresses with this role could pause the `TaikoL1` contract, stopping the proposal and proving of L2 blocks as well as direct deposit of ETH to L2. The role was replaced by the `chain_watchdog` in phase 3, which can pause the `TaikoL1` and the `Bridge` contracts.

- `snapshooter`: Any address with this role could trigger snapshots on the `TaikoToken`. This role was removed in phase 3 of the audit.
- `withdrawer`: Addresses with this role can sweep any token sent to the `TaikoL2` contract. Note that the `TaikoL2` contract is not intended to hold the user's funds.
- `rollup_watchdog`: The rollup watchdog can remove instances from the SGX verifier contract to stop previously trusted SGX instances from being able to submit SGX proofs. This role was replaced by the `sgx_watchdog` role in phase 3.

Each of these roles presents a unique set of permissions in the Taiko protocol.

Security Model and Trust Assumptions

There are several trust assumptions for users of the protocol:

- All of the addresses with the roles mentioned above can impact users in different ways and some, if compromised, could freeze or steal users' funds.
- The nature of the cross-chain communication protocol used by Taiko assumes that all the nodes in the network composed of the different Taiko-connected chains are honest. If a chain is compromised and can pass arbitrary signals, including arbitrary state roots for another chain, assets held by the vaults could be stolen. It is important for users to monitor which chains are added to this network, and to understand the implications of connecting to another chain.
- Because all the contracts rely on the address resolver and manager contracts to authenticate each other, it is important that they stay synchronized and share the same address manager across the different contracts on a chain. Loss of synchronization could have unexpected effects, including loss of user funds.
- All of the contracts inherit from `EssentialContract` and can thus be paused and upgraded by the owner. We assume this role is given to the DAO.
- All the tokens deployed to L2 have pausability and upgradability. In phases 1 and 2 of this audit, such tokens had snapshot and voting capabilities as well. Users should be aware of this fact as it could impact how tokens can be used on L2.
- While the Taiko L2 is Ethereum equivalent, some things are different than on L1. For example, there is no guarantee as to the time between L2 blocks and the difficulty opcode is computed differently. Developers are assumed to be aware of these differences when deploying to Taiko so they can ensure their contracts remain secure.

- The genesis state root is assumed to not include any backdoor.
- A quorum of guardians can execute arbitrary state transitions on the rollup. If enough guardians are compromised or dishonest, this may result in a loss of funds for users.
- The destination owner of a cross-chain message is assumed to be able to receive funds and to be an address controlled by message senders. Users should understand that wrongly setting this address could result in loss of funds.
- It is assumed that when the `ERC20Vault` owner changes the bridged token contract of a canonical token, the new bridged token matches the previous version in name, symbol, and decimals, and is fully operable as intended by the `ERC20Vault`. Furthermore, the owner is expected to do so only after all the users have migrated to the latest contract before the change.
- It is assumed that if a bridged token migration happens from or towards a token which does not support the `IBridgedERC20Migratable` interface, the Taiko DAO would do their diligence to ensure that no user funds are lost.
- It is assumed that the quotas of the `QuotaManager` are only increased over time, as a decrease can interfere with ongoing asset bridging.
- The soundness of the state transition proofs associated with blocks relies on the assumption that third party provers are incentivized to contest invalid state transitions. This incentivization in practice depends on the relative prices of the Taiko token and of proving a block. It is possible for the price of the Taiko token to be too low to incentivize third party contesters to contest an invalid state transitions. It is assumed that if such a situation was judged likely, the Taiko DAO would step in to increase the liveness and contest bonds.

The above security considerations and trust assumptions are inherent to the design of the codebase. Thus, we assume in the report below that the aforementioned actors are not compromised and behave honestly.

Critical Severity

C-01 Vulnerability in Block Proposal Function Allows Token Theft - Phase 1

The `proposeBlock` function is designed to enable the proposing of L2 blocks to an L1 chain, requiring proposers to indicate a prover responsible for proving the block. The process needs proposers to provide a signature that verifies the prover's agreement to specific terms, such as fee costs, block metadata hash, and the token used for fee payment. Anyone can be a prover as long as they can execute the necessary computations for proving the L2 block and have 250 TKO or more tokens that could ensure their commitment to proving blocks within a certain timeframe.

A security flaw is present within the `AssignmentHook`'s `onBlockProposed` method. This method, which is executed during the block proposal transaction, manages the transfer of fees from the proposer to the prover. If the transaction involves fee payment in a non-ETH token, the contract utilizes the `safeTransferFrom` method to facilitate the fee transfer from the Coinbase address to the prover's address. However, a malicious proposer can set the Coinbase address to be a user's address that has previously granted an ERC-20 token allowance to the `TaikoL1` contract (usually a prover's address as they could set the maximum allowance on its TKO tokens to participate as a prover). By doing so, the proposer can specify the fee amount to be equivalent to the entire token allowance, choose that token for the payment of the fees, and assign a controlled prover to be the one that receives the tokens.

Consider adjusting the fee payment logic to ensure that the Coinbase address is not used when transferring the fees. In addition, consider using an address that correctly represents the `proposeBlock` caller, preventing unauthorized redirection of ERC-20 token allowances.

Update: *Resolved in pull request #16327 at commit 7423ffa. A new variable was added to track the block proposer, who now pays the fees.*

C-02 Bridge Signals Can Be Forged to Drain the Protocol - Phase 3

Cross-chain messages can be sent by calling the `sendMessage` function on the `Bridge` contract. This function in turn calls `sendSignal` on the `SignalService` contract with the hash of the sent message. When a message is processed on the destination chain, `_message.value + _message.fee` is checked not to exceed the current quota. If it exceeds the quota, the message's status is set to `RETRIABLE` and can be re-executed later. Otherwise, the message is validated not to call forbidden addresses and to match `onMessageInvocation`'s function selector, after which the external call is made to the destination address.

However, the checks constraining the target of the message or the function selector are consequently not applied when a message is retried. Assuming an ETH quota on L1, an attacker can thus perform the following actions to invoke any message on L2:

1. Send a message on L2 that will exceed the available L1 ETH quota, targeting the `SignalService` to call `sendSignal`. The sent signal must be the hash of the crafted message, which the attacker will process on L2 in step 5.
2. The attacker attempts to process this message on L1, but as the message exceeds the quota, it would be directly set to `RETRIABLE` without validating its target or the function selector.
3. This message can then be retried and will make the `Bridge` contract invoke the call to `sendSignal`. Note that `sendSignal` is non-payable, so the message in step 1 would need to have a value of zero but a fee in excess of the available L1 ETH quota.
4. The L1 signal will be synchronized with L2 through normal protocol operation.
5. The attacker can process the crafted message.

Sending arbitrary messages from the `Bridge` is a powerful attack vector as it also allows spoofing any address through the `Bridge` context. This makes it possible to forward arbitrary amounts of ETH or mint any ERC-20, ERC-721, or ERC-1155 token on the L2. These tokens can then be bridged back to drain user assets from the `Bridge` and vault contracts on L1, subject to quotas per amount of time given by the `QuotaManager`. Furthermore, by spoofing the `realOwner` of the `DelegateOwner` contract, a backdoor can be installed that will be exploited at a later point in time.

Note that we assumed above for simplicity, but without loss of generality, that there were only two layers. This is because the exploit can occur between any two layers where one layer has an ETH quota defined.

Consider validating that messages are not calling arbitrary functions or forbidden addresses before setting their status to `RETRIABLE` or when retrying the message.

Update: *Resolved in pull request #17411 at commit 304aec2. The invocation target and function signature are now also checked in the* `retryMessage` *function. However, the fix introduced a double spending attack vector of the fee that was addressed in pull request #17446 at commit 891967d.*

High Severity

H-01 Initialization Script of TaikoL2 Allows Reinitialization of the Rollup - Phase 1

The `TaikoL2` contract is intended to be initialized by a script setting the state variables in the genesis state root directly instead of calling the `init` function. However, this script does not initialize the `_initialized` variable to `1` on the rollup contract. This means that `init` could be called after deployment during the first block by a malicious actor to set itself as the owner of `TaikoL2` or change the address manager, giving it control of critical functions.

Once granted the owner role, a malicious actor could, for example, withdraw any token sent to the `TaikoL2` contract, or change the base fee of the rollup at will by manipulating the configuration. An example of malicious usage of this privilege could be to include one transaction setting the base fee to 1 wei at the beginning of its blocks before resetting it to a very high level at the end, allowing this user to have a monopoly over L2 transaction inclusion by pricing out all the other block proposers as well as circumventing any gas limit.

Consider modifying the genesis script to initialize all the variables of the `TaikoL2` contract correctly, as well as adding a test scenario to ensure the storage layouts of the L2 contracts initialized by calling `init` are the same as the ones obtained by running the initialization script. This would also be useful for rollup users to be able to reproduce the genesis state root and ensure that no backdoor was included in the initial state.

Note that the initialization script is out of the scope of this engagement.

Update: *Resolved in pull request #16543 at commit 37fa853. The deployment script was modified to initialize those variables.*

H-02 Incorrect Basefee Calculation on Taiko Rollups - Phase 1

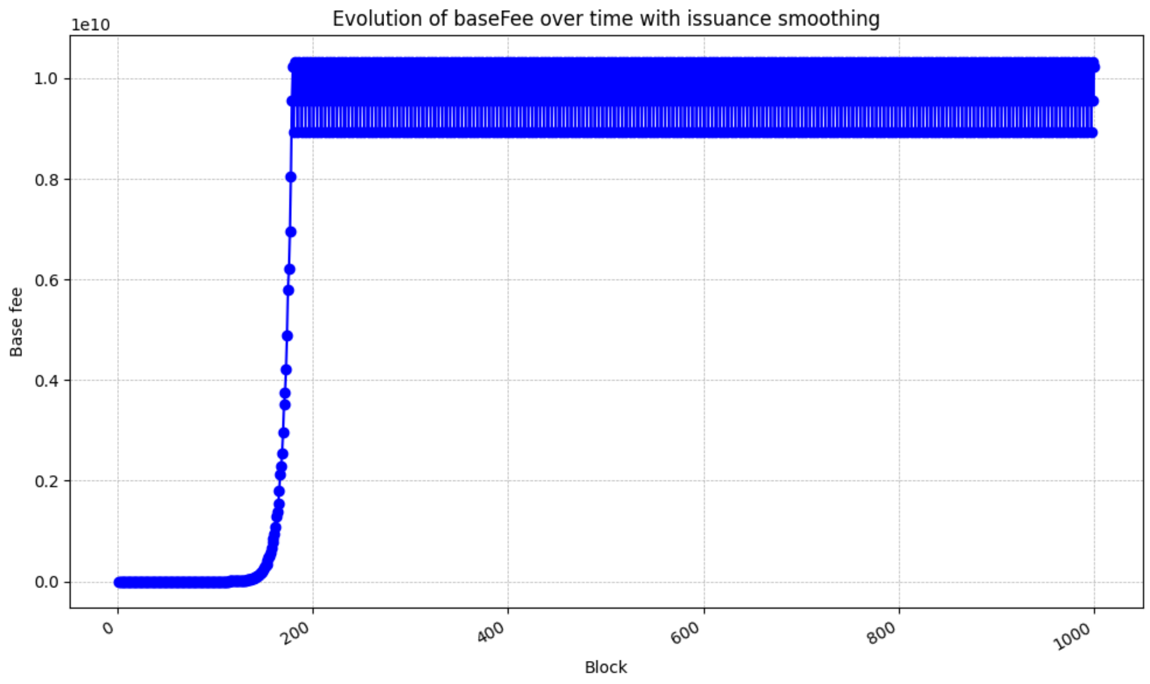
On a Taiko rollup, the basefee at block `n` is computed as `b(n) = b0 * 1/TA * exp(gas_excess / TA)`, where `b0 = 1`, `T` the gas target is `60` million gas and `A` is the adjustment factor set to `8`. This formula is explained as originating from the research around making EIP-1559 more like an AMM curve. However, the formula used does not match the AMM formula, in which `basefee` would be computed as `(eth_qty(excess_gas_issued + gas_in_block) - eth_qty(excess_gas_issued)) / gas_in_block`. The base fee is actually computed as the derivative of the AMM curve at the last `gas_excess` reached in the previous transaction. This makes the computation closer to exponential EIP-1559, except the formula to compute the base fee is multiplied by a factor `1 / TA`.

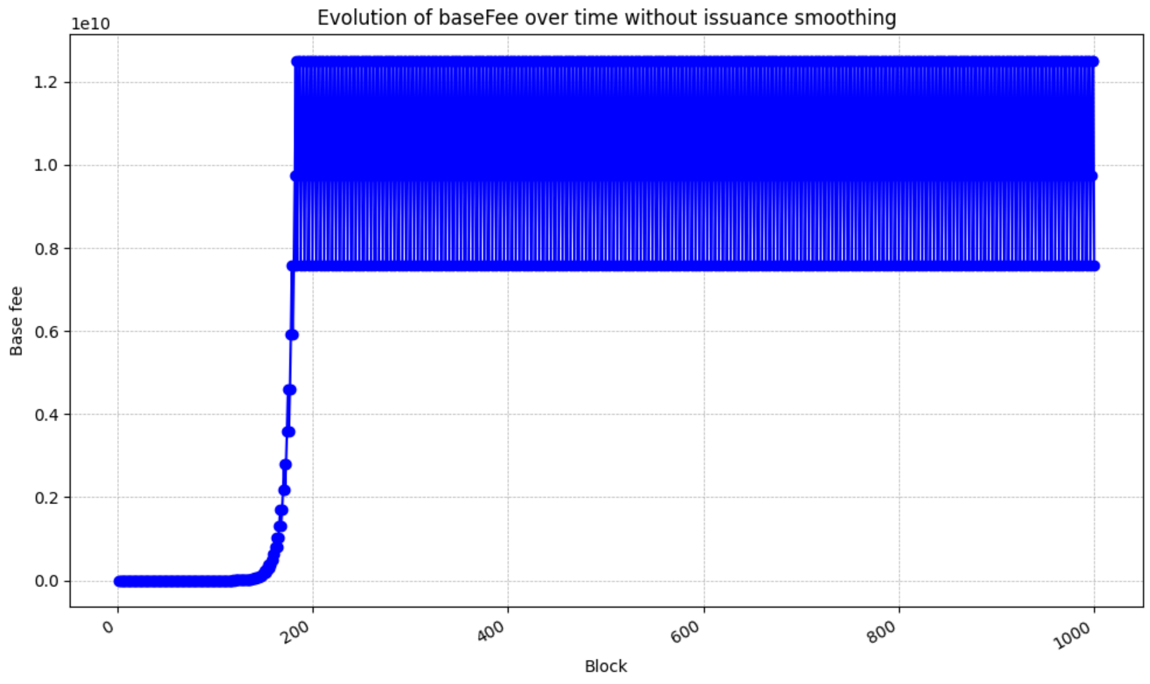
There are multiple issues with the EIP-1559 implementation:

1. The `lastSyncedBlock` state variable is updated every 6 blocks, but the new issuance of gas every L2 block is computed as `(_l1BlockId - lastSyncedBlock) * gasTargetPerL1Block`. This means that there will be significantly more L2 gas issuance than expected due to the fact that `lastSyncedBlock` is not updated every L1 block. For example, let us assume that `lastSyncedBlock = 1000` and the current L1 block number is `1002`. Let us now assume that we want to anchor any number of new L2 blocks. The new issuance of L2 gas for such a block would be `(_l1BlockId - lastSyncedBlock) * gasTargetPerL1Block = 2 * gasTargetPerL1Block = 120` million, which is higher than the block gas limit. This would result in the base fee staying at 1 wei no matter how many L2 blocks are created, or how full they are. In practice, this would mean that no base fee is collected at all, and we expect the system to revert to a first-price auction mechanism for the pricing of its gas. Additionally, this means that there is no limit to the amount of L2 blocks which can be proposed or to the gas which can be consumed per unit of time. As long as the compensation earned from tips by block proposers is enough to pay for the proposal of new blocks, blocks could be proposed without any limit on network congestion. This could also have unexpected consequences related to the size of the ring buffers, as well as the equilibrium between the proving and proposing of blocks. Consider issuing new L2 gas only on new L1 blocks.

2. The issuance of new L2 gas is currently done every 6 L1 blocks, and could be done every L1 block as recommended in 1). However, issuing gas in one chunk every >= 4 L2 blocks runs counter to one of the goals of EIP-1559, which is to act as a gas smoothing mechanism. This creates undesirable spikes in the base fee, where the next L2 block anchored following an L1 block is significantly cheaper than the following blocks.

Consider smoothing the L2 gas issuance over multiple blocks (e.g., by accumulating the L2 gas issuance in a storage variable and spreading it over 4 L2 blocks). The figures given below as examples assume that new gas issuance occurs every 4 L2 blocks and compare issuing gas in one chunk with issuing it over 4 L2 blocks. The demand for block space is modeled as being infinite below 10 gwei and 0 above.





3. As mentioned above, the current implementation of EIP-1559 is similar to an exponential 1559 where `b0 = 1/TA = 2.08e-9`. Since `b0` is the minimum value that the base fee

can take, its minimum value should be the minimum possible base fee (i.e., 1 wei instead of 2.08e-9 wei which is scaled up to 1 wei). This would avoid having to compensate for the low `b0` by adjusting the excess gas in the exponential upward, which reduces the range before the input to the exponential gets too big and loses some precision. Consider removing the division by `T * A` and setting `b0` to 1 wei as done in Ethereum (see [1] [2]).

Consider addressing the above issues to improve the gas pricing mechanism on L2. In addition, consider adding documentation around the gas logic as well as the lack of constraints on L2 block times.

Update: *Partially resolved at commit a981ccd. All the points in the issue except the second one were addressed. The Taiko team stated:*

> *Regarding the base-fee smoothing concept, we've already implemented and tested it, confirming its functionality. However, we've decided to first deploy a simpler version in production to assess its behaviors before integrating the base-fee smoothing feature. This decision aligns with potential future changes, including possibly eliminating the anchor transaction framework in favor of handling all anchor-related logic (including EIP1559 calculation) on the client-side and in the prover, to simplify the protocol further.*

H-03 Block Submission Susceptible to DOS - Phase 1

When proposing a new block, the block is added to a ring buffer. If too many blocks have been proposed since the last verified one (i.e., if the ring buffer is close to being "filled") additional block proposals revert. This mechanism is there to avoid erasing blocks that have been proposed but not yet verified from the buffer. Once a block has been proposed, it can be proven. Assuming block proposals are uniform over L1 blocks, 99.9% of them will require to be proven with at least an SGX proof on mainnet. Once proven and after a cooldown period of 1 day has passed, a block can then be verified. During this cooldown period, block proofs can be contested. For SGX-proven blocks, contesting with a proof of the same tier does not require a valid proof and resets the timestamp associated with the transaction. Doing so thus resets the cooling period before the contested block can be verified.

However, this makes it possible for a malicious user to DoS block submission by sequentially contesting blocks with SGX proof as their cooldown period expires. For example, let us assume that 42 blocks have been proposed and proven using SGX, but have not yet been verified. A malicious user can proceed as follows:

- Contest the first block with an SGX proof, which requires paying a contest bond of 500 TKO tokens. Note that such a contest does not require a valid SGX proof. This stops the block verification process during the cooldown period of 24 hours, but blocks continue being proposed and accumulated to the ring buffer. Assuming a new L2 block is proposed every 3 seconds, the proposed block ring buffer gets filled up to ~1/42 of its size during the cooldown.
- After the 24-hour cooldown window for the first block has elapsed, the malicious user can then contest the second block in the same way. This again costs 500 TKO tokens and freezes block verification for 24 hours, during which blocks continue being proposed and accumulated in the buffer.
- The malicious user proceeds as above and continues contesting blocks sequentially as the cooldown periods expire.

At the end of this process, the ring buffer is now filled and proposing new blocks is impossible. The malicious user can continue this process, paying 500 TKO tokens for each additional day of DoS.

We note that this attack is easy to detect and that it is possible for the guardians to prove the affected transitions with the highest tier to reduce the cooldown window for verification to one hour, increasing the cost of such an attack. The total cost without guardian intervention is around `42 * 500 = 21000` TKO tokens, and the highest possible cost assuming guardian intervention is around `42 * 500 * 24` = 504,000 TKO tokens, which is high but not unreasonable for a motivated attacker. Effective intervention entails guardians being on the lookout and sending at least five approval transactions on L1 every hour, which is error-prone and costly. The impact of any error in this process could also be dramatic as guardians cannot re-prove invalid transitions (see M-01).

The impact of stopping the rollup block proposals is unknown. For example, it could be increased by DeFi activity emerging on Taiko as lending markets would not be able to process oracle price updates, which could provide an incentive to a malicious actor to pause the rollup. In any case, user funds would be stuck.

Consider preventing the contestation of transitions once the cooldown window has expired. Alternatively, consider monitoring for such situations and adapting the `contestBond` or the `cooldownWindow` to make such attacks uneconomical.

***Update:** Resolved in pull request #16543 at commit 37fa853. Transitions can no longer be contested after the cooldown window has expired, except if a higher level proof is provided.*

Medium Severity

M-01 The Guardian Cannot Re-Prove Invalid Transitions - Phase 1

The highest proof tier is the guardian tier, allowing a guardian address controlled by the security council to effectively prove any state transition. Should the guardian make a mistake, the intention of the code is for the guardian to be able to re-prove a different transition. However, when re-proving a different transition for the highest tier, a check is made to ensure that the `contestBond` of the transition is equal to 0. This check makes any attempt to re-prove a transition revert, as the `contestBond` would have been set to 1 in the previous transition proof.

This revert makes it impossible for anyone to correct an invalid transition when proven once by the guardian. If such an occurrence were to happen, the only practical solution would be to upgrade the implementation before the 1-hour cooling window expires to avoid the invalid transition being verified.

Consider replacing this check by `ts.contestBond == 1`, as well as adding a corresponding unit test of the ability for the guardian to re-prove transitions to the test suite.

Update: *Resolved in pull request #16543 at commit 37fa853.*

M-02 Governor Proposal Creation Is Vulnerable to Front Running in OpenZeppelin Contracts - Phase 1

The Taiko governance employs the `GovernorCompatibilityBravoUpgradeable` contract from OpenZeppelin version 4.8.2 which is susceptible to a front running attack. By front running the creation of a proposal, an attacker can take control as the proposer and have the option to cancel it. This vulnerability can be leveraged to indefinitely prevent the submission of legitimate proposals, posing a significant risk to the governance process.

Consider updating the OpenZeppelin contracts library to version 4.9.1 or later as these versions contain the patch for this issue. This will enhance the security and integrity of the governor proposal mechanism.

Update: *Resolved in [pull request #16360](#) at commit [2a0fe95](#). The Taiko team stated:*

> *Thank you for reporting this bug. We have upgraded @openzeppelin library to "4.9.6".*

M-03 Design Flaw in Bridge Contract Suspension Mechanism - Phase 1

The Bridge contract implements a [`suspendMessages`](#) function that can be invoked by either the owner or a designated watchdog. This function is intended to suspend or unsuspend a message invocation by updating the `receivedAt` field within the message's proof receipt. Specifically, suspending a message sets this field to the maximum possible value for a `uint64`, whereas unsuspending it assigns the current timestamp. This approach introduces a vulnerability as the `receivedAt` field also serves to indicate whether a message has been proven, with any value greater than zero being interpreted as such. Consequently, the ability to suspend or unsuspend unproven messages inadvertently marks them as proven due to the non-zero `receivedAt` value.

One significant implication of this flaw is the potential exploitation by the owner or watchdog. By unsuspending a message and manipulating its data or value to match the bridge's balance or vaults' tokens balances, they could trigger the [`recallMessage`](#) method, effectively draining all assets from the bridge and the vaults. On a Taiko rollup, the attack can be executed immediately as there is no invocation delay, and for Ethereum, the attacker needs to wait only one hour. This could be even more problematic if the attacker does it with a bridged token on L2, as they could mint an infinite amount of tokens and affect Dapps deployed on the L2.

To mitigate this risk, consider introducing a separate dedicated field within the proof's receipt structure explicitly for saving the value prior to the suspension. Alternatively, to efficiently manage the `receivedAt` timestamp during suspension and unsuspension events, you can consider adopting a reversible computation method (e.g., implementing `receivedAt = 2 ** 64 - 1 - receivedAt` offers a pragmatic approach). This method ensures that upon suspension or unsuspension, the `receivedAt` value is transformed in a manner that allows for the original timestamp to be recovered through the same operation.

Update: *Resolved in [pull request #16545](#) at commit [c879124](#). The Taiko team stated:*

> *Thank you for identifying this issue. We've implemented a fix to ensure that even if the bridge_watchdog behaves maliciously, it cannot falsely mark a message as received without proper verification of its delivery.*

M-04 Incomplete Role Renunciation in `TaikoTimelockController` Deployment - Phase 2

The deployment script `DeployOnL1` is used by the Taiko team to set up the contracts on the L1 chain. This includes the `TaikoTimelockController`, which serves as the owner of the other contracts. An oversight occurs during the deployment process: the `TIMELOCK_ADMIN_ROLE`, which can bypass the timelock delay, is not renounced by the deployer post-deployment.

Initially, the deployment script sets the zero address as the owner in the `init` function, but the actual owner defaults to `msg.sender`, who also receives the `TIMELOCK_ADMIN_ROLE`. Although the script revokes the rest of the roles from `msg.sender` after the deployment, the admin role is not revoked. There is a revoke call targeting `address(this)`, but this contract was never assigned the `TIMELOCK_ADMIN_ROLE`. This means that `msg.sender` is still an admin of the timelock after deployment.

To mitigate this risk and enhance transparency, consider incorporating a `renounceRole` call at the end of the deployment script, allowing the deployer to explicitly renounce the `TIMELOCK_ADMIN_ROLE`.

***Update:** Resolved in pull request #16751 at commit abd18e8.*

M-05 Some ERC-20 and ERC-721 Tokens Can Not Be Bridged - Phase 2

ERC-20 tokens can be bridged through the `ERC20Vault` contract by calling the `sendToken` function. If the token is canonical to this side of the bridge, its metadata is fetched and transmitted to the other side. This metadata is fetched through external static calls. The same is done for ERC-721 tokens.

However, metadata are defined as optional in the ERC-20 and ERC-721 standards. Additionally, some tokens such as `MKR` do not respect the standard and return their name and symbol as a `bytes32` instead of a `string`. The current implementation would revert and prevent such ERC-20 and ERC-721 tokens from being bridged. Moreover, tokens with empty `symbol` and `name` are permitted to bridge, but the deployment of the `BridgedERC20` contract would fail on the other side of the bridge and the message would have to be recalled.

Consider enclosing the calls to fetch the metadata in `try/catch` blocks or turning them into low-level calls, and assigning a default value on failure. Additionally, consider adding support for `bytes32` decoding, and assigning a default value on invalid or empty decodings. Examples of such decoding functions can be found here and here.

***Update:** Resolved at commit dd8725f. The `symbol` and `name` can now be `bytes32`, and they are assigned default values if they do not exist. The `BridgedERC20` deployments no longer revert if these properties are empty.*

M-06 Cross-Chain Owner Cannot Call Privileged Functions - Phase 2

The `CrossChainOwned` abstract contract can be inherited to allow an owner on a different chain to execute privileged actions on the child contract. For example, the `TaikoL2` contract inherits from `CrossChainOwned`. When a `CrossChainOwned` contract is called by the bridge, the cross-chain sender and the original chain are validated to correspond to the cross-chain owner, after which an external call is made to itself.

However, it is impossible for a cross-chain owner to use this mechanism to call functions protected by `onlyOwner` on `TaikoL2`. This is because the `_owner` variable of the contract has to match the address of the cross-chain owner, but the cross-chain owner calling the contract results in the contract calling itself with an external call. This means that calls to any function protected by `onlyOwner` would revert as the `msg.sender` would be `address(this)` and not the cross-chain owner. This makes it impossible for the cross-chain owner to call functions protected by `onlyOwner` on the `TaikoL2` contract, such as `withdraw` or `setConfigAndExcess`.

Consider allowing the cross-chain owner to call privileged functions.

***Update:** Resolved at commit 37fa853. The Taiko team stated:*

> *Cross-Chain owner got removed (or reworked) and the new is "DelegateOwner", which will have the same role, to act like the owner essentially, for contracts deployed on L2.*

M-07 Security of 'SignalService' Network Is No Higher Than Its Weakest Node - Phase 2

The `SignalService` contract handles cross-chain communications. Sending a cross-chain message sets a storage slot in the local `SignalService`, and the state of a chain can be synchronized to another chain by calling the `syncChainData` function. This state synchronization is for example done during L2 block verification on L1 or in the anchor transaction at the beginning of new blocks on L2. Once synchronized, this allows the destination chain to prove that a signal has been sent from the source chain by doing a Merkle proof against this chain's state. These proofs are also allowed to be recursive by allowing for multiple "hops", meaning that L3 -> L1 communication can be done in a single call by proving that the L3 state was stored to L2 and that the L2 state was stored to L1. To save gas in future calls, the intermediate states can be cached locally when such a proof is done.

This design effectively creates a network of interconnected chains, where each one is trusted and can communicate with any other. However, in practice, it is likely different chains will have very different trust and security assumptions.

For example, there could be two Taiko L2s on top of Ethereum: A canonical one and another one (called "friendly fork") originally vetted by the canonical Taiko DAO. A Taiko L3 could then settle to this friendly fork and have their `SignalService` be connected to the L3, for example by social engineering and/or by promising them financial incentives (eg. using their token as gas, etc.). While the canonical Taiko L2 is owned by the canonical chain's DAO, the Taiko L3 could be owned by a multisig for security purposes. In such a situation, this multisig which has not been vetted by the canonical Taiko L2 community could steal assets from the canonical rollup. Note that direct malicious hop proofs are impossible since the address resolution of `SignalService` does offer some protection: the L3 address can not be resolved without the canonical DAO's agreement. However, this protection is insufficient, and here is how such an attack could be done:

1. The compromised multisig uses the L3 to synchronize an invalid state root from the friendly fork. This invalid state root contains a malicious message from the friendly fork's L2 vault to the L1 vault asking for a withdrawal of all the assets in the L1 vault, including those belonging to the canonical chain.
2. The L3 state root is synchronized to the friendly fork.
3. `proveSignalReceived` is called on the friendly fork, with a hop proof that the malicious message was signaled in the friendly fork's invalid state root, which was signaled in the L3 state root, which was synchronized to the friendly fork in step 2). Note that this is possible as multi-hop proofs are allowed to have the same `chainId` multiple times, and the first hop is allowed to have the `chainId` of the current chain. The goal of this step is to cause the malicious friendly fork's state root to be cached and signaled by the friendly fork itself.
4. The friendly fork's state root is synchronized to the L1.
5. `processMessage` is called on the L1 vault with a proof that the malicious message belongs to the invalid L2 state which was signaled as part of the L2 state, which was

itself synchronized in step 4). This could for example be used to drain all the assets from the L1 vaults.

The current design is dangerous as the security of all the rollups in the network depends on its weakest node: No matter how decentralized the canonical rollup gets, if any centralized chain's `SignalService` is added to the network by anyone else, this centralized actor has the power to drain the other rollup's assets.

This concern with the current design could be alleviated in multiple ways, for example:

- Hop proofs should be prevented from containing the same `chainId` multiple times.
- A specification should be written detailing what trust and security assumptions are required for a new chain to be added to the network and ratified by the DAO as well as any chain added to the network. Any new chain added to the network would need to be approved by the canonical DAO, and any violation of these conditions would result in expulsion from the network.
- The caching mechanism could be deactivated.
- If there is no use case for it, `proveSignalReceived` could revert if `_chainId` is the same as `block.chainid` as done in intermediary hops.

Update: *Resolved at commit dd8725f. Different hops can no longer have the same `chainid`, and the first hop can no longer use the current chain's `block.chainid`.*

M-08 Quota Manager Can Cause Recalled Funds to Be Stuck - Phase 3

ETH and ERC-20 tokens can be sent through the `ERC20Vault` and `Bridge` contracts, respectively. When funds are sent from these contracts to the users, `consumeQuota` is called on the `QuotaManager` contract.

However, the quota is not checked when sending funds from the users to the contracts. This makes it possible for a user to send funds, mark the message as failed on the destination chain, but not be able to recall their message and claim their funds back on the source chain [1] [2] if the amount exceeds the maximum quota. Hence, these funds would be stuck until the quota is increased or removed.

Consider validating during message sending that the funds sent are lower than the current quota to avoid issues when recalling messages.

Update: *Acknowledged, not resolved. The likelihood of this issue justifies not introducing additional code changes. However, assets that have a quota are recommended to be monitored to check for the scenario depicted above. The Taiko team stated:*

> *Quota only applies to tje ERC20 vault and we believe for ETH and tokens whose quota is configured to be non-zero (USDC, USDT, TKO), it is very unlikely the claiming will fail thus the message be marked as failed on the destination chain.*

M-09 Reached Quota Can Cause Loss of Fee - Phase 3

The `Bridge` contract calls `consumeQuota` on the `QuotaManager` contract when sending ETH. This notably happens during message processing. If the message is processed by its `destOwner` and the maximum quota is reached, the message is set to retriable. The message can then either be retried or set as failed and recalled on the source chain. However, `message.fee` is lost in such a situation as it is not refunded to the `msg.sender`.

Consider adding a condition to return the fee to the `destOwner` if the fee is non-zero and does not exceed the available quota. In case `message.fee` exceeds the available quota, consider reverting to protect users from losing their fee. Note that this would mean any message with a fee exceeding the quota would be stuck until the quota is raised. In addition, and to mitigate such issues, consider having the same quotas on L1 and L2, and validating when sending a message that the value sent (inclusive of the fee) does not exceed the quota. As quotas are only expected to increase over time, it would then be possible to simply revert when processing a message which exceeds the quota.

Update: *Resolved in pull request #17411 by refunding the value and fee in the* `retryMessage` *function. However, this fix was followed-up with pull request #17446, where* `processMessage` *reverts when the quota is reached or (a part of) the fee is refunded. If the message goes into retriable state and is unable to be invoked, the value is also refunded.*

Low Severity

L-01 Assigned Prover Signature Is Underspecified - Phase 1

The `AssignmentHook` contract is a canonical hook that can be used by proposers during block proposals. The proposer sends to the hook a signature that was given by a prover that contains information about the proving fees as well as the proposed block. This, for example, protects provers against being paid a smaller fee to prove a bigger block than agreed upon. These signatures can either come from EOAs or contracts using EIP-1271 signatures.

However, the current signature scheme does not include the assigned prover address, which allows for signatures to be valid across several smart wallets sharing an owner. Let us assume, for example, that Alice owns two smart wallets, `A` and `B`, and expects the assigned prover to be wallet `A`. Then, wallet `B` can be used as the assigned prover, since the signature would indeed originate from Alice and does not constrain it to being used only with wallet `A`. While already done by some smart contract wallets, consider including the user address - in this case the assigned prover address - in the signature to avoid potential issues. An example of such a vulnerability, allowing signature replay across smart wallets can be found here.

Furthermore, there is no way for a prover to cut an exclusive deal with a block proposer. While the data signed includes the `metaHash` and thus the `block.coinbase` address, the block could be proposed by a different address than the one designated as `block.coinbase`. Provers may want to have a simple mechanism to prove any block proposed by a designated address.

Consider including the assigned prover in the `AssignmentHook` signed data to avoid potential issues with smart wallets. Additionally, consider including the address of the block proposer in the data signed by the provers.

Update: *Resolved in pull request #16665 at commit 2b27477.*

L-02 The Snapshooter Cannot Take Snapshots of the Taiko Token - Phase 1

The Taiko token, designed with snapshot capabilities through inheritance from the `ERC20SnapshotUpgradeable` contract of the OpenZeppelin library, has an initialization flaw.

This token uses a `snapshot` method, designed to give both the token's owner and a designated snapshooter the ability to take snapshots. However, an initialization error within the `EssentialContract` — a contract defining the `onlyFromOwnerOrNamed` modifier to ensure exclusive access for the owner and snapshooter to a method — affects the functionality.

Specifically, the `EssentialContract` attempts to manage its initialization through two distinct versions of the `__Essential_init` initializer. The first variant establishes ownership and sets the paused state to `false`, whereas the second variant, in addition to executing the tasks of the first, sets the `addressManager` for the resolver. The issue arises from the fact that the `TaikoToken` contract uses the former variant, which neglects defining the `addressManager`, thereby making resolutions of the snapshooter's address return the zero address. This oversight blocks the snapshooter from performing their intended role, as the absence of a valid `addressManager` setting cannot be rectified post-initialization, forcing a contract upgrade for resolution.

Consider using the `__Essential_init` initializer that properly incorporates the `addressManager`. This change ensures the system's architecture aligns with its designed capabilities, enabling the snapshooter to fulfill their role in snapshot management.

Update: *Resolved in pull request #16394 at commit c64ec19.*

L-03 Inefficient Feature Inclusion in `EssentialContract` - Phase 1

The `EssentialContract` serves a critical role across various contracts in the protocol by inheriting features such as pausability, reentrancy protection, and access to an address manager. However, not all inheriting contracts fully utilize these features. This design does not align with the Solid Principle of single responsibility which advocates for a system design that promotes separation of concerns for efficiency and reduced error likelihood.

To enhance modularity and adherence to the single responsibility principle, consider splitting the `EssentialContract` into distinct contracts by responsibility. This approach enables contracts to only inherit the necessary features, thereby streamlining the codebase and minimizing unnecessary complexity.

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

> *Thank you for your feedback. We have decided not to adopt the suggested approach, as we believe the associated risks are minimal and manageable.*

L-04 L2 Block Difficulty Is Susceptible to Manipulation - Phase 1

When proposing an L2 block on L1, the `difficulty` associated with the block is computed as the hash of `(block.prevrandao, numBlocks, block.number)`. This difficulty is used to compute the minimum tier associated with proving the block and is also used on L2 as the value returned when calling `block.prevrandao`. Block proposers can propose multiple blocks in one transaction using blobs.

However, the `difficulty` can be biased by the proposer by proposing multiple blocks. For example, this means that proposers can predict before the block proposal if proving the block will require a zk proof or an SGX proof. If the cost difference between zk and SGX proving warrants it, block proposers could in the same transaction submit multiple empty blocks, and include L2 transactions only when the difficulty is such that an SGX proof is required, effectively only using SGX to prove non-empty blocks and partially avoiding the minimum tier mechanism.

Similarly, depending on how the `block.prevrandao` is used by contract developers on L2, this could incentivize block proposers to bias the difficulty at the time of L2 transaction inclusion. While each L1 validator can have one bit of influence over `block.prevrandao` on L1, the ability to atomically propose multiple L2 blocks and choose in which to include L2 transactions give them more control over the L2 `block.prevrandao` than may be assumed by application developers.

Consider monitoring proposers for abuses of their privilege when proposing multiple blocks and choosing where to include transactions. In practice, the profitability of such strategies will depend on the fixed costs in gas associated with block proposals compared to the cost of zk proofs. If this issue was observed in practice, potential solutions could include the use of a VRF oracle, or disabling the ability for proposers to propose multiple blocks in one transaction. No matter which option is chosen, consider documenting any control block proposers may have over `block.prevrandao` on L2, as this is important for application developers.

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

> *Internally we have been back and forth regarding how to decide a block's min tier. There is no perfect solution without major protocol design changes. The current decision is that we keep this part of the code as-is, at least for now.*

L-05 Uncontrolled Gas Consumption When Sending ETH in Cross-Chain Messages - Phase 1

Cross-chain messages can be sent through the `Bridge` contract by calling the `sendMessage` function on one side and `processMessage` on the other. Messages can include a fee to incentivize relayers to process messages automatically on the destination chain. As the gas limit is specified in the message, relayers can simulate in advance to know if processing a message is profitable.

However, when sending the funds to the target address on the destination chain, all the gas is forwarded. This is because calling `sendEther` on an address without specifying a gas amount forwards all the gas. In practice, this would make it harder for relayers to estimate the gas costs associated with processing a message.

Consider only passing a fixed amount of gas when forwarding the refund to the refund address to make it easier for relayers to estimate the maximum amount of gas consumed when processing a message.

Update: *Resolved in pull request #16666 at commit 4909782.*

L-06 Honest Provers Can Lose Money - Phase 1

When proposing a block, a prover is assigned to prove it. This assigned prover is the only one who can prove the first transition of this block for the duration of the proving window, in exchange for providing a liveness bond. This liveness bond can be lost if the proven state transition is later proven invalid by a higher level proof, but is returned if the transaction was not contested.

However, it is possible for an honest prover to lose TKO tokens when assigned to a block if another malicious actor is ready to lose more. For example, let us name Henry an honest prover, Mallory a malicious user, and look at the following sequence of events:

1. Henry is assigned to prove a block, and sends 250 TKO as liveness bond to the `TaikoL1` contract. He correctly proves the state transition with an SGX proof, providing another 250 TKO as validity bond.
2. Mallory contests the state transition, providing 500 TKO as contest bond to the `TaikoL1` contract.
3. In the same transaction, Mallory re-proves the block and confirms Henry's transition using a zk proof. In doing so, Henry earns `1/4 * contestBond = 125` TKO, and

gets his original validity bond of 250 TKO back. Mallory gets `1/4 * contestBond = 125` TKO as she is the new prover. She provides a new validity bond of 500 TKO.
4. The block is verified, and Mallory as the final prover gets back half of the liveness bond as well as her validity bond, `1/2*250 + 500 = 625` TKO in total.

As a result of the above, Henry has lost his initial liveness bond but got 1/4th of the contest bond, a net loss of `125` TKO. Mallory has lost 3/4th of her contest bond but got back half of the liveness bond, a net loss of `250` TKO.

While Mallory lost more than Henry, Henry was honest but still lost money. If Mallory is malicious, has a lot of TKO tokens and wants to establish herself as the sole prover of Taiko blocks, she could for example systematically contest any SGX transition on a block she is not an assigned prover of, in order to disincentivize anyone but herself to prove blocks. This could be used in an attempt to get a monopoly on block proving. Alternatively, she could attempt to contest any SGX proof in order to disincentivize blocks from being proven at all.

Note that the scenario above can be mitigated if Henry initially provides a zk proof, as the guardian has the option of returning the liveness bond to the assigned prover when resolving a contest. Such a defense strategy however would only work for zk proofs and would require constant interventions from the guardian.

Consider adding a mechanism to return the liveness bond to the assigned prover in scenarios where a block has been contested but the assigned prover's proposed transition was valid.

Update: *Resolved at commit dd8725f. The liveness bond is now returned directly to the assigned prover when the block is proven within the proving window, or if the guardian intervenes and decides to return the liveness bond.*

L-07 Duplicated USDC Bridging - Phase 2

The `sendToken` function can be called to bridge an ERC-20 token, represented as another token on the destination chain. Initially, native USDC on Ethereum L1 will be represented as a BridgedERC20 on Taiko L2s. Later, if a native USDC is deployed on a Taiko L2, it is possible to migrate to this native USDC version by migrating the `BridgedERC20` version of USDC through a `USDCAdapter` contract.

However, this configuration by default makes it possible to bridge USDC from L2 to L1 in two different ways. The first one is to bridge the `USDCAdapter`, which after the migration would bridge to the native USDC on L1. The second way is to bridge the new native USDC on Taiko L2 directly as it is a canonical token, which would thus bridge to a newly deployed

`BridgedERC20` on L1. This dual-path approach risks confusing users and fragmenting USDC liquidity across Taiko's rollup ecosystem.

Consider keeping only one way to bridge USDC after the migration. This could be achieved by blacklisting bridging native Taiko USDC token directly to force users to bridge through the `USDCAdapter`.

***Update:** Resolved at commit dd8725f. USDC and `BridgedERC20` tokens now use a common interface, so the `USDCAdapter` was removed.*

L-08 `ERC20Airdrop` Claims Can Be DOSed - Phase 2

The `ERC20Airdrop` contract allows users to claim tokens by providing a Merkle proof of their airdrop. To do so, users can call the `claimAndDelegate` function and have to provide airdrop information as well as a signature to delegate their tokens.

However, the transaction calling `claimAndDelegate` is susceptible to a frontrunning attack where an adversary could independently use the user's signature with the `ERC20Votes` token. This action consumes the nonce and causes the user's original transaction to revert, thereby blocking them from claiming their tokens.

Consider making it possible for users to claim without delegating, for example by adding a second function or by wrapping the `delegateBySig` in a `try/catch` clause.

***Update:** Resolved in pull request #16738. The Taiko team stated:*

> *Removed the delegation part.*

L-09 `TimelockTokenPool` Signatures Can Be Replayed - Phase 2

The `TimelockTokenPool` contract will be deployed multiple times to allocate grants to investors, team members and other grantees. The status of granted tokens goes from locked to owned to unlocked over time. Once unlocked, the amount can be claimed by calling the `withdraw` function. One of the two `withdraw` functions takes as arguments a destination address and a signature. The signed data is validated to correspond to `keccak256(abi.encodePacked("Withdraw unlocked Taiko token to: ", _to))`.

However, such signatures can be replayed to claim tokens for anyone else at any time, causing multiple potential issues. For example, tokens claimed for investors or team members could cause unexpected and increased tax obligations. Additionally, since claiming tokens has a fixed cost per claimed token, this could be used to force the recipient to lose money if the price of the claimed tokens is lower than the cost paid. Note that since the `TimelockTokenPool` contract will be deployed multiple times on one chain, signatures could be replayed across several of these contracts if they share a common grantee address. If the timelock is intended to be deployed across multiple chains, signatures could also in theory be replayed across chains.

Consider preventing signature replay across time, for example by adding a nonce. Additionally, consider adding the contract address to the signed data to avoid signature replay across multiple instances of the `TimelockTokenPool` contract, as well as adding the `chainid` if it is intended to be deployed across multiple chains. If the data is intended to be signed through third-party frontends, EIP-712 could also be integrated.

Update: *Resolved in pull request #16934. The Taiko team stated:*

> *Ackowledged and removing the whole contract. The new ones are under supplementary-contracts package (in the repo) so not part of protocol anymore. Unlocking is simplified and no signature based withdrawals.*

L-10 `evaluatePoint` Precompile Calls Revert - Phase 2

The `evaluatePoint` function is used to validate that the opening of a KZG Commitment (corresponding to a blob in practice) at a point `x` is equal to the purported `y` by calling Ethereum's point evaluation precompile. This precompile expects its commitment and proof arguments to be 48 bytes each.

However, the `_commitment` and `_pointProof` arguments are of type `bytes1[48]`, and passed by calling `abi.encodePacked`. The packed encoding of a `bytes1[48]` is done in-place without the length but with each element padded to 32 bytes. This means that the commitment and proof arguments sent to the precompile are `32 * 48 = 1536` instead of 48 bytes long, and thus the first padded 3 bytes of the commitment are erroneously interpreted as representing both the commitment and the proof. Because these are padded and mostly 0s, they are unlikely to correspond to a valid commitment/proof pair.

In practice, this means that any normal call to the point evaluation precompile would revert. Additionally, gas could be saved by changing the type of these arguments to `bytes` and validating their length separately. We note that while the `Lib4844` library is in the scope of this audit, it is not currently used by the contracts in scope.

Consider changing the `_commitment` and `_pointProof` arguments to be of type `bytes`. Additionally, if the `Lib4844` is intended to be used in practice, consider adding tests targeting this library.

Update: *Resolved in pull request #16969. The Taiko team stated:*

> *This file has been deleted.*

L-11 Vaults Are Not ERC-165 Compliant - Phase 2

Token vaults inherit from the `BaseVault` contract. This contract implements the `IERC165` interface and exposes the `supportsInterface` function.

However, ERC-165 states that ERC_165 compliant interfaces have to return `true` when `interfaceID` is `0x01ffc9a7` (the EIP-165 interface ID). Following the steps from ERC-165 to detect if the vaults implement ERC-165 would thus return false.

Consider adding the ERC-165 interface ID to the supported interfaces.

Update: *Resolved in pull request #16935.*

L-12 Token Migrations Can Cause Losses - Phase 2

Tokens inheriting from the `BridgedERC20Base` contract can be migrated by the owner or by calling the `changeMigrationStatus` function of the `ERC20Vault`. If a migration is triggered, token owners can burn their tokens to mint the same amount of tokens on the new contract. If the migration is triggered by the bridge, old tokens are added to a blacklist and have to be migrated before being able to bridge.

However, there are multiple migration scenarios that could cause accidental user losses:

- If a new migration is triggered before some users have migrated from a previous migration, then these users would effectively lose their tokens as they would no longer be able to call `mint()` on the new contract nor bridge their tokens.
- If a migration is triggered by the owner instead of the `ERC20Vault`, the vault would no longer be able to call `burn` or `mint` on the migrated token. This would prevent users

from bridging and could cause losses as the new migrated token would not be bridging to the same contract when bridging through the vault.
- If a migration happens to a token that doesn't implement the `IBridgedERC20` interface, user losses could happen as well.

If such concerns are shared, consider addressing the above instances to protect users' funds from errors during migrations. Potential solutions include adding a delay to migrations during which no new migration can be done, preventing the owner from calling `changeMigrationStatus`, and validating that the new token implements the `IBridgedERC20` using ERC-165 during migrations.

***Update:** Resolved at commit dd8725f. A minimal delay of 90 days between migrations was added to give users time to migrate. Additionally, the owner is now prevented from triggering migrations. We note that the new token is not enforced to support `IBridgedERC20Migratable`, as this would prevent migrating from a bridged token to a canonical implementation on Taiko. Migration of such token would have to be handled by the DAO to avoid loss of funds.*

L-13 Unexpected Features in BridgedERC20 Tokens - Phase 2

When an ERC20 token is bridged from a source chain to a destination chain via the `ERC20Vault`, it results in the creation of a `BridgedERC20` token. The `BridgedERC20` implementation currently includes extra functionalities such as voting, snapshot capabilities, upgradability, and pausability, which may not be present in the original canonical token. This discrepancy could lead to confusion as users might not expect these additional features in the bridged token representation.

To address this issue, consider adopting a simpler `BridgedERC20` implementation that retains only the essential features of an ERC20 token or to clearly documenting the additional functionalities. Clear documentation will ensure that users are fully informed about the capabilities of the bridged tokens, preventing any potential misunderstanding.

***Update:** Partially resolved in pull request #16950. The `BridgedERC20` token contract no longer offers the vote/checkpoint feature. Remaining features like upgradability and pausability should be thoroughly documented, both within the contract itself and in external resources.*

L-14 Banned Addresses Could Be Called By Cross-Chain Messages - Phase 2

The status of messages sent to addresses in the `addressBanned` mapping of the `Bridge` contract is set to `DONE`. Otherwise, the message can be retried if it fails.

However, the check for banned addresses is not made when retrying a message. This makes it theoretically possible for a message to call a newly banned address if its status was set to `RETRIABLE` before the address got added to the `addressBanned` mapping. This could for example be exploited if future deployment addresses are predictable and the code is open source, by calling such functions before the contract is deployed and its address is banned. This would set up dormant messages which can be processed later.

Consider checking the target of messages in the `retryMessage` function to avoid potential issues.

Update: *Resolved in pull request #16394 at commit a6470a1. The Taiko team stated:*

> *Not only because of this but also to avoid confusing, we removed the banning address feature completely.*

L-15 Some Bridged Messages Can Not Be Refunded - Phase 2

When processing a message on its destination chain's `Bridge` contract, a refund can be emitted if the target address was "banned". This refund is always sent to the `refundTo` or the `destOwner` address, even if this refund is 0. If the refund is unsuccessful, the call to `processMessage` reverts. Otherwise, if the refund and the message call were successful, the status of the message is set to `DONE` or `RETRIABLE`. A message with the `RETRIABLE` status can then be retried and its status set to `FAILED` if desired. This allows for the message to be recalled on the source chain so its original sender can be refunded.

However, this makes it possible for messages whose `refundTo` or `destOwner` address do not accept ETH to always revert when processed on the destination chain, making the message not recallable. As such messages can never be recalled, the sender would have lost funds. Note that because the refund is always sent even with a value of 0, this could happen even if the destination address is not banned. This could for example happen if `refundTo` is a contract without a `receive` or a `fallback` function.

Consider not calling `sendEther` when the refund sent is 0. This would prevent the above from occurring except when the target address is banned, in addition to saving gas. Alternatively and to account for this last case if desired, consider setting the status of messages targeting banned addressed to `FAILED` instead of `DONE`. This would allow removing the concept of refunds entirely, simplifying the code, and allowing for these messages to be recalled by their original sender.

Update: *Partially resolved at commit dd8725f. The ban of addresses was removed, and message processing now uses* `sendEtherAndVerify` *which does not trigger an external call if the value sent is 0. We note that it is however still possible for funds to get stuck if the* `destOwner` *is a contract without payable* `fallback` *and* `receive` *functions.*

L-16 'DelegateOwner' Does Not Check for Contract Existence - Phase 3

The `DelegateOwner` contract is intended to be deployed on L2 and set as the owner of all the other L2 contracts. The DAO on L1 can then call it through the `Bridge` contract to execute arbitrary calls in its name. This could, for example, be used by the DAO to execute privileged functions on L2 from L1. Calls from the `DelegateOwner` can be either low-level calls or delegate calls based on an input parameter.

However, low-level calls in Solidity do not check for contract existence. Such calls could thus be considered successful if the contract called has not been deployed yet, resulting in silent failures.

Consider validating the contract's existence if the given `call.txdata` is non-empty.

Update: *Resolved in pull request #17328 at commit 39505e5 and pull request #17480 at commit 60d5d22.*

L-17 Storage Collision in Bridge Contract - Phase 3

The `Bridge` implementation was upgraded. The previous implementation stored `nextMessageId` as a `uint128` in storage slot 251. The new implementation stores two `uint64` values instead, `__reserved1` and `nextMessageId`, declared in that order.

However, this creates a storage conflict as packed storage slots are filled right to left and lower-order aligned. This means that a `nextMessageId` with value 10 in the old version would upgrade to a `__reserved1` of 10 and a `nextMessageId` of 0. While `__reserved1` is set back to 0 in the `init2` function, the `nextMessageId` would restart from 0. We note that because the `Bridge` contract is disabled on the mainnet, it is impossible for users to increment `nextMessageId`. It is thus unlikely to cause issues in practice.

Consider validating that `nextMessageId` was not incremented before upgrading the contract to avoid loss of funds.

Update: *Resolved. The Taiko team stated:*

> *This issue affects the testnet since the previous version of the code was deployed. Fortunately, no one exploited this bug to manipulate the bridge and steal testnet tokens.*
>
> *We now have a script to automatically verify that storage layouts are compatible with the previous version. On the testnet, we will avoid upgrades that could cause these kinds of issues.*

L-18 Inexplicit Revert - Phase 3

When a token is bridged through the `ERC20Vault` contract, the destination chain message value is derived as `msg.value - _op.fee`. However, this can lead to an underflow revert if the provided fee exceeds the message value.

Consider checking the values beforehand, as done in the `Bridge` contract.

Update: *Resolved in pull request #17329 at commit f8042a2.*

L-19 Unrestricted Receive Function - Phase 3

The `Bridge` contract implements a `receive` function that allows anyone to send ETH to the contract. This might be necessary during setup to meet the amount of ETH that was minted in the genesis state on L2 and is in circulation on the rollup. However, for users interacting with the bridge, it can be a pitfall to accidentally get their funds locked.

Consider restricting the `receive` function to the owner role, or finding another way to balance these amounts between L1 and L2.

Update: *Resolved in pull request #17330 at commit 4ef2847. The Taiko team stated:*

> *Currently our approach is to remove the receive, but only after the genesis, once we seeded the initial liquidity.*

L-20 Lack of Constraints During Migration - Phase 3

The `changeBridgedToken` function of the `ERC20Vault` contract can be called to change the representation of a canonical token on the current chain. While the function can only be called by its owner, consider adding the following validations to reduce the risk of error:

- A validation that `ctoken.addr != 0` and `_ctoken.chainid != block.chainid`.
- If desired, a check could be added that the `_btokenNew` address has code. This would constrain new canonical token deployments to always happen before migrations, but would reduce the risk of error.

Update: *Resolved in pull request #17333 at commit 8d14e84.*

L-21 Incorrect Calldata Gas Accounting in Message Processing - Phase 3

The minimum gas limit set for a message accounts for the calldata costs of an encoded `Message` struct when processing the message. This is done by adding `(message.data.length + 256) / 16` to a flat gas reserve.

The following issues were identified:

- The computation of gas cost per calldata byte is done by dividing, although it should be multiplying by 16.
- The word length accounted for the encoded `Message` struct is 7 words plus the dynamic data length, while additionally overcharging one word to account for the rounded up dynamic size. The actual encoding of the Message struct requires 13 words plus the dynamic data length, while the dynamic length can be rounded up to a multiple of 32. Hence, the bytes length formula should be `13 * 32 + ((dataLength + 31) / 32) * 32`.
- The calldata cost is not accounted for in the gas charged when calculating the fee, although it is at the expense of the processor. This is due to `gasCharged` only accounting for the gas consumption between the two calls to `gasleft()`, which does not include the gas costs associated with encoding `_message` in the transaction calldata.

Consider correcting the minimal gas and fee calculations to reflect more accurate costs.

Update: *Resolved in pull request #17284 at commit 1a5b040 and pull request #17529 at commit 8c91db2.*

L-22 Message Processors Can Control Gas and Fee Consumption - Phase 3

The `Bridge` contract allows users to send cross-chain messages between chains connected by a `SignalService`. When sending a cross-chain message, users can input a gas limit and a fee associated with the message. If the gas limit is non-zero, third party message processors can call `processMessage` on the destination chain to process the message in exchange for a part of the fee depending on the consumed amount of gas.

In pseudocode, and assuming no precision issues, a message processor is paid:

```
min(
    msg.fee,
    gasCharged * msg.fee / msg.gasLimit,
    gasCharged * (msg.fee / msg.gasLimit + block.basefee) / 2
)
```

A rational, profit-maximizing message processor will want to minimize their expense (`block.basefee * gasCharged`) while maximizing their revenue (`msg.fee / msg.gasLimit * gasCharged`). Assuming a rational message processor, messages will only be processed if the profits are positive, i.e., if `msg.fee / msg.gasLimit` is greater than `block.basefee` (the third `min` case). However, as this profit scales with the gas charged, the processor can make the processing message transaction more gas expensive to get the maximum profit by extracting the full `msg.fee` as revenue.

To control how much gas is charged, the message processor can add an arbitrary number of zero bytes at the end of the encoding of the `HopProof[]` in `_proof`. These excess bytes will be ignored during abi decoding but will consume unnecessary gas by expanding memory at the expense of the original message sender.

Because each zero byte of calldata costs 4 gas to the message processor and the memory expansion happens in the proxy contract as well, such attacks can only be profitable if `message.gasLimit` and/or `message.fee` are very high. A proof of concept shows that this exploit requires rather extreme conditions which are unlikely to occur in practice.

If such attacks are a concern, consider adding a validation that `proof.length` has a size in bytes below a certain threshold (e.g., 200,000). The quadratic part of the memory expansion costs only matters here for very high `proof.length`. Alternatively, consider removing the manual decoding of `hopProofs` and replacing it with a `HopProof[] calldata` argument.

Update: *Resolved in pull request #17429 at commit 2fd442a.*

L-23 Inaccurate Gas Limit on Message Invocation - Phase 3

Users can send cross-chain messages by calling `sendMessage` on the `Bridge` contract. An optional gas limit can be defined for this message. In that case, if a user wants their message to be executed with at least `A` gas, `message.gasLimit` has to be at least `A + minGas`, where `minGas` depends on the message length plus a flat gas amount.

When a message is processed by a third party, the gas remaining before invoking the user's message is validated to be greater than `64 / 63 * A`. This is because of EIP-150, which silently caps the amount of gas sent in external calls to `63 / 64 * gasleft()`. If such a check were not present, it would be possible for the gas to be capped by EIP-150, executing the call with less than `A` gas.

However, while the intention was correct, a gas limit of `A` is not guaranteed. This is due to the gas expenses that accumulate between the EIP-150 check and the actual message execution on the target:

- Memory expansion costs
- Account access
- Transfers
- Opcode costs

Because `1 / 64 * gasleft()` has to be enough to execute the rest of the `processMessage` function without running out of gas, a gas limit below `A` on the target is only possible for high gas limits. This proof of concept shows the issue in practice.

Consider addressing the above to build a more predictable bridge for users and projects sending cross-chain messages. What follows is an example of how this could be achieved.

The goal of the computation is to ensure that EIP-150 does not silently cap the amount of gas sent with the external call to be less than `A`. We note:

- `memory_cost`: the amount of gas needed to expand the memory when storing the inputs and outputs of the external call.

- `access_gas_cost`: the gas cost of accessing the `message.to` account. This currently corresponds to 2600 gas if the account is cold, and 100 otherwise.
- `transfer_gas_cost`: the cost of transferring a non-zero `msg.value`. This cost is currently 9000 gas but provides a 2300 gas stipend to the called contract.
- `create_gas_cost`: the cost of creating a new account, currently 25000 gas. This only applies if `message.value != 0`, `message.to.nounce == 0`, `message.to.code == b""`, and `message.to.balance == 0`. Since `message.to` is checked to have code, this cost can be ignored here.

Thus, we want to check that the following:

```
63 / 64 * (gasleft() - memory_cost - access_gas_cost - transfer_gas_cost)
```

is at least as much as `A` (reference implementation). The sum of `access_gas_cost` and `transfer_gas_cost` can be upper-bounded with `2_600 + 9_000 - 2_300 = 9_300`. The `memory_cost` is cumbersome to compute in practice, but by estimating the costs through tests, we can upper-bound the cost of memory expansion for up to `10_000` bytes of `message.data` in the context of the call with the formula `1_200 + 3 * message.data.length / 32`.

As such, it would be possible to validate that the call will have enough gas by checking the following condition right before the external call is made:

```
63 * gasleft() >= 64 * A + 63 * (9_300 + 1_200 + 3 * message.data.length / 32) +
small_buffer
```

This would be accurate for messages with up to `10_000` bytes of `message.data`. Any message above this limit could be required to have a gas limit of zero which would force it to be processed by its `destOwner`. A similar approach, without a constraint on the data size, has been adopted by Optimism and can be used as inspiration.

Update: *Resolved in pull request #17529 at commit a937ec5. After further discussions with the audit team, the fix implemented follows an alternative, more efficient approach than the one suggested in the issue.*

Notes & Additional Information

N-01 Inconsistent Use of Named Returns - Phase 1

Throughout the codebase, there are multiple instances where contracts have inconsistent usage of named returns in their functions:

- In the `Bridge` contract
- In the `DevnetTierProvider` contract
- In the `MainnetTierProvider` contract
- In the `TaikoL1` contract
- In the `TaikoL2` contract
- In the `TestnetTierProvider` contract
- In the `ERC1155Vault` contract
- In the `ERC20Vault` contract
- In the `ERC721Vault` contract
- In the `SgxVerifier` contract
- In the `SignalService` contract
- In the `TimelockTokenPool` contract

Consider being consistent with the use of named returns throughout the codebase.

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

> *Decided not to proceed with a fix.*

N-02 Lack of Indexed Event Parameters - Phase 1

Throughout the codebase, several events could benefit from having indexed parameters:

- The `GuardiansUpdated` event of `Guardians.sol` could index the `version` parameter.
- The `MessageSuspended` event of `IBridge.sol` could index the `msgHash` to be consistent with the other events defined above.
- The `Anchored` event of `TaikoL2.sol` could index the `parentHash`.

- The `MigrationStatusChanged` event of `BridgedERC20Base.sol` could index both the `addr` and the `inbound` parameters.
- The `Withdrawn` event of `ERC20Airdrop2.sol` could index the `user` parameter.
- The `SignalSent` event of `ISignalService.sol` could index the `app` and the `signal` parameters.
- The `Claimed` event of `MerkleClaimable.sol` could index the `hash`.
- The `BlobCached` event could index the `blobHash` parameter.

To improve the ability of off-chain services to search and filter for specific events, consider indexing the event parameters identified above.

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

> *We decided not to add additional indexes as most of these events are not used directly by end users.*

N-03 Signatures Do Not Use EIP-712 - Phase 1

The `AssignmentHook` is a canonical hook using signatures to validate that the prover agreed to prove a certain block in exchange for a certain amount of fees. These signatures can be from EOAs or from smart contracts using EIP-1271. However, the signed data does not respect EIP-712. Depending on how the prover is expected to sign this data (e.g., through a third-party frontend) it could be beneficial to use a standard supported by the service providers such as wallets, etc.

Consider formatting the signed data according to EIP-712.

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

> *We decided not to proceed with this as Assignments are signed by provers in the backend.*

N-04 Gas Optimization - Phase 1

The following opportunities for gas optimizations were found:

- The `EssentialContract` contract packs two variables, `__reentry` and `__paused`, as two `uint8` values in one storage slot. However, by default, these variables are neither read nor written at the same time. Consider changing them to be `uint256` to save gas.

- Resolved names such as "proposer", "tier_provider", and "taiko_token", domain separators such as "PROVER_ASSIGNMENT" or "SIGNAL", and verifier names such as `"tier_sgx"` could be `constant bytes32` contract variables.
- During during block verification, the `tko` token address could be cached, similar to the tier provider.
- In the `GuardianProver` contract:
  - The `Approved` and the `GuardianApproval` events are always emitted consecutively and have some duplicate information. Consider only emitting one.
  - The indexation of the guardian could use the loop iterator `i + 1` instead of `guardians.length` to avoid reading from storage in the loop.
  - The `minGuardians` storage variable read in the `for` loop during approvals could be cached to memory, similar to `guardians.length`.
  - The `address[] memory _newGuardians` argument could be stored in calldata to avoid copying the array to memory.
- In the `sendEther` function, no returned value is necessary and the allocation of 64 bytes of memory could be avoided.
- In the `anchor` function, the `blockhash(parentId)` and the state variable `gasExcess` could be cached to memory to save gas during the `Anchored` event emission.
- In the `onBlockProposed` function of hooks, the `_blk`, `_meta` and `_data` arguments could be in calldata rather than memory by default to save gas.

Consider updating the identified instances to save gas during the operation/deployment of the protocol.

Update: *Partially resolved at commit 84f06f3. All items except items one and four were addressed.*

N-05 Inefficient Bridge Message Handling - Phase 1

When a message has been processed before and the message call has failed, bridge users are currently forced to retry a message invocation on the destination chain to transition its status to `FAILED`. This is the case even in situations where it is known in advance that `_invokeMessageCall` will again not be successful. This status transition is necessary for users to invoke the recall function to recover their funds on the source chain.

Consider adding a way for users to directly transition a message's status from `RETRIABLE` to `FAILED` without having to retry executing the message call. This would reduce gas consumption for affected users and enhance their overall user experience by streamlining the process.

Update: *Resolved in pull request #16669 at commit dce651e.*

N-06 Code Quality and Readability Suggestions - Phase 1

The following opportunities to improve the quality of the codebase were identified:

- The `graffiti` component of the `Transition` struct is unused and could be removed to save gas. Alternatively, consider documenting its use.
- The `TransitionState` struct could be renamed to `StateTransition`.
- The `_createTransition` function could be renamed (e.g., to "`_fetchOrCreateTransition`") as it does not always create a new transition.
- The config validity check for the `blockMaxProposals` parameter could check that "`blockMaxProposals <= 1`" instead of `== 1`.
- `_operationId` could be renamed to `_blockId` for consistency and clarity. Similarly, the `proofSubmitted` event argument could be renamed to `minGuardiansReached` as the `Guardians` contract has no context on a proof.
- The `init` function of the `TaikoTimelockController` does not grant the `PROPOSER_ROLE` nor the `EXECUTOR_ROLE` to the `TaikoGovernor` contract. While the `TIMELOCK_ADMIN_ROLE` is granted to the owner, meaning that it could grant these roles separately, it would be clearer and simpler if these were granted during initialization. Consider passing the address of the `TaikoGovernor` to grant these roles during initialization. Alternatively, consider documenting that these roles are to be granted separately.
- The `_genesisBlockHash` could be checked to be nonzero during initialization to ensure the following transitions can be proven.
- The `L1_TOO_MANY_TIERS` error is unused and could be removed.
- The `AM_INVALID_PARAMS` and `AM_UNSUPPORTED` error names are vague and could be changed to be more accurate (e.g., to "`AM_ADDRESS_ALREADY_SET`" and "`AM_PAUSE_UNSUPPORTED`", respectively).

Consider addressing the above instances to improve the clarity and safety of the codebase.

Update: *Partially resolved in pull request #16667 at commit 250ad7d. All the issues were addressed except for the second point. Note: The TaikoTimelockController was removed in pull request #16933.*

N-07 Missing or Misleading Documentation - Phase 1

The following opportunities to improve the clarity of the documentation were found:

- The structs defined in the `TaikoData` library would benefit from some added documentation on what the different fields represent.
- The `EthDeposit` struct is incorrectly documented as using 1 slot but actually uses 2 slots as it is 40 bytes long.
- `meta.coinbase` is documented as being the L2 block proposer, but it is only chosen by the block proposer.
- In the `proveBlock` function, a comment mentions that if the transition does not exist the `tid` will be set to 0. However, it is set to 1 in practice.
- A conditional branch for the `TIER_OP` tier is present in the code but could be documented as only being used for testnet.
- The contest bond is documented as being burned from the prover, but it is only transferred and can in fact be regained if the transition is proven to be invalid.
- The `_overrideWithHigherProof` function would benefit from better documentation overall (e.g., when `ts.contester == 0` for the first transition, there is technically no existing contest but the code still uses this branch).
- The NatSpec around the `getBasefee` function indicates that the function can be used to "get the basefee and gas excess [...]". However, the function only returns the base fee.
- The NatSpec around the `__ctx` variable in the `Bridge` contract indicates that it fits in 3 slots, whereas it only occupies 2.
- The NatSpec around the `proveMessageReceived` function is incorrect as it was duplicated from the function above it.
- The NatSpec around the `getInvocationDelays` function has an extra "and" in the phrase "a message can be executed since and the time it was received".
- The `processDeposits` function could use some additional documentation (e.g., around the fee logic).
- The `Message` struct in the `IBridge` interface could document that the `gasLimit` of a message can be set to 0 so that only the `destOwner` can process the message on the destination chain. Additionally, the `gasLimit` could be documented as not being respected on message retries or when processed by the owner.
- A comment around the `gasLimit` is reversed, as the remaining gas is used if called by the owner.

Consider addressing the above instances to improve the clarity and readability of the codebase.

Update: *Partially resolved in pull request #16681 at commit f31a6ac. All the mentioned instances were fixed except for item 8 in the list. Note that the fifth item was resolved separately in commit 2c63cb0.*

N-08 Typographical Errors - Phase 1

The following typographical errors were identified in the codebase:

- `isAssignedPover` should be `isAssignedProver`.
- `deleys` should be `delays`.
- `Indenitifer` should be `Identifier`.
- "send" should be "sent".
- "choose use" should be "choose to use".
- `tran` should be `transitions`.

Consider addressing the above instances to improve the quality of the codebase.

Update: *Resolved in pull request #16667 at commit 250ad7d.*

N-09 Changing Governor Parameters Requires an Upgrade - Phase 1

The `TaikoGovernor` contract is intended to be set as the proposer and executor of the `TaikoTimelock` contract, which will be set as the owner of the other deployed contracts. Parameters of the `TaikoGovernor` contracts include the `votingDelay`, the `votingPeriod`, and the `proposalThreshold`.

These parameters are hardcoded in the contract and would require a contract upgrade if there was a need to update them.

If such flexibility is desired, consider inheriting from `GovernorSettingsUpgradeable` to allow for these parameters to be changed by governance without having to upgrade the contract.

Update: *Resolved in pull request #16687 at commit eba82ba.*

N-10 Unused Named Return Variables - Phase 1

Named return variables are a way to declare variables that are meant to be used within a function's body for the purpose of being returned as that function's output. They are an alternative to explicit in-line `return` statements.

In the codebase, there are instances of unused named return variables:

- The `invocationDelay_` return variable in the `getInvocationDelays` function in `Bridge.sol`
- The `invocationExtraDelay_` return variable in the `getInvocationDelays` function in `Bridge.sol`

Consider either using or removing any unused named return variables.

Update: *Resolved in pull request #16600 at commit f6efe97.*

N-11 The `TimelockTokenPool` May Not Be Compatible With ERC-4337 - Phase 2

ERC-4337 is a standard that allows smart contracts to behave like user accounts. Under the EIP there will be ERC-4337 smart contract accounts in addition to Externally Owned Accounts (EOA). The former account type is incompatible with some code, such as `tx.origin` or `ecrecover`.

The `TimelockTokenPool` uses `ecrecover` for the withdrawal of the grant, which could be inconvenient for smart wallets and multisigs.

If smart wallet support is desired, consider using EIP-1271 to allow smart contract accounts to verify signatures instead of `ecrecover`. This could for example by achieved by using the `SignatureChecker` library instead of `ECDSA`.

Update: *Resolved in pull request #16934. The Taiko team stated:*

> *Removed (underway) the TimelockTokenPool, and the new one (under supplementary-contracts) will always be EOAs.*

N-12 Typographical Errors - Phase 2

The following typographical errors were identified in the codebase:

- "authrized" [1] [2] should be "authorized"
- "Indenitifer" [1] [2] should be "Identifier"
- "converison" should be "conversion"

Consider correcting the above instances to improve the quality of the codebase.

Update: *Resolved in the following commits: 810e723, 0eec494 and 1b9ba53.*

N-13 Gas Optimizations - Phase 2

The following opportunities for gas optimizations were found:

- When emitting the `MigratedTo` event, the `msg.sender` could be used in place of the `migratingAddress` variable to save gas.
- The `isClaimed` mapping in the `MerkleClaimable` contract could be made more efficient. Consider using a BitMaps instead, for example see Uniswap's `MerkleDistributor`.
- The `ERC1155Vault` contract could call the `_burnBatch` (for example by implementing a public `burnBatch` function in `BridgedERC1155`) and `safeBatchTransferFrom` functions of ERC-1155 instead of the `burn` and `safeTransferFrom` functions. This would reduce the number of external calls and reduce gas costs.

Consider updating the identified instances to save gas during the operation/deployment of the protocol.

Update: *Partially resolved at commit dd8725f. All the items were resolved except the second one. The Taiko team stated:*

> *As the claiming will be done on our L2, gas cost shall not be an issue, and would want to avoid such changes as the infrastructure (for claiming and putting together off-chain data) is already in place.*

N-14 Missing or Misleading Documentation - Phase 2

The following opportunities to improve the clarity of the documentation were found:

- The `CrossChainOwned` abstract contract allows an "owner to be a local address or one that lives on another chain". However, during initialization the `_ownerChainId` is checked to be different from `block.chainid`, and it is used to synchronize the state root with another chain. It is thus unclear which value to give `_ownerChainId` if the owner is "local".
- The `BridgeTransferOp` struct is documented in the `BaseNFTVault` contract but not in the `ERC20Vault`. Consider adding this documentation.
- The NatSpec around `blockSyncThreshold` says that "a value of zero disables syncing", but this is not the case. Consider removing this comment.
- The phrase "uniquely from chainId, kind, and data" above `isChainDataSynced` was copied from above and should be removed.
- The documentation around the `sendSignal` function explains that it "sets the storage slot to a value of 1", however in practice the storage slot is set to `signal`.
- The `getMyGrantSummary` function computes the cost to withdraw a token grant. Because the grant amount is divided by `1e18` before being multiplied by the cost, it is possible for the `costToWithdraw` to be off by one token over the lifetime of the grant. Consider documenting this to make it clear to readers. Besides, the arguments and return values of the `getMyGrantSummary` function are not documented.

Consider addressing the above instances to improve the clarity of the codebase.

***Update:** Resolved in pull request #17483. The `CrossChainOwned` and `TimelockTokenPool` contracts have been removed, so the first and last items no longer apply.*

N-15 Lack of Indexed Event Parameters - Phase 2

Multiple events could benefit from having indexed parameters:

- The `InstanceAdded` event could index its `replaced` parameter.
- The `Withdrawn` event could index its `to` parameter.

To improve the ability of off-chain services to search and filter for specific events, consider indexing the event parameters identified above.

Update: *Resolved in pull request #16949. The Taiko team stated:*

> *TimneLockTokenPool is deleted, but the other one is a good spot.*

N-16 Code Quality and Readability Suggestions - Phase 2

The following opportunities to improve the codebase were identified:

- The `SafeCast` import in the `SignalService` contract is unused and could be removed.
- The `Strings` import in the `BridgedERC20` contract is unused and could be removed.
- The `customConfig` state variable in the `TaikoL2EIP1559Configurable` contract is not used. Additionally, it is not initialized alongside the other variables. Consider removing it or initializing it during initialization.
- The initializer `__Essential_init` is not protected by a `onlyInitializing` modifier. Consider adding one for consistency.
- The `__gap` variable of the `TimelockTokenPool` contract is defined as a `uint128[]`, which is error-prone. Consider changing it to be a `uint256[]` for consistency and safety.
- The same `MigratedTo` event is used in two different contexts: when migrating from, and when migrating to. Consider emitting the `migratingInbound` variable as part of the `MigratedTo` event to differentiate these two situations.
- The `MessageRetried` event could include the new status of the message.
- The `EssentialContract` could have a public `__reentry` getter. This could be useful for integrations wanting to avoid potential read-only reentrancies when reading from the state of the protocol.
- The `BaseVault` contract could support the `IMessageInvocable` ERC-165 interface id.
- The deployment of bridged token contracts by vaults [1] [2] [3] currently uses `CREATE`. If desired, it could use `CREATE2` with a salt depending on `ctoken.addr` and `ctoken.chainId` for example. This would make the deployment address of bridged tokens standardized and predictable.
- The `validSender(address _app)` modifier validates the `_app` address is nonzero, and could thus be renamed to "nonZeroApp" for consistency and clarity.
- `MerkleClaimable` is an abstract contract allowing an owner to set a Merkle root against which proofs can be submitted during claims. The ability for an owner to set a

new Merkle root even as an airdrop is already happening should be prevented or documented to minimize the risk of error and user loss.
- Consider crediting Optimism explicitly in the NatSpec for the `RLPWriter` and `RLPReader` libraries. This would make it easier to review and diff against their version, in addition to crediting them for the libraries.
- `msg.sender` could be replaced by `user` in the `_handleMessage` function for clarity and consistency.

Consider addressing the above instances to make the code clearer and safer.

Update: *Partially resolved at commit dd8725f. All the items listed were addressed except for the tenth and twelfth ones.*

N-17 `BridgedERC1155` Does Not Return Correct URI - Phase 2

The `BridgedERC1155` token currently does not override the `uri` method to return URIs specific to each token ID. As it stands, the method returns a uniform URI regardless of the input token ID, leading to a bad user experience.

Consider overriding the `uri` method such that it dynamically generates or retrieves URIs based on the token ID provided. Implementing this change will align the token functionality with typical expectations for ERC1155 tokens and enhance user experience.

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

> *It was a design choice picked by Daniel and Brecht. So basically to give a signal, that this particular NFT is a bridged one, and the source could be find here and there (with .buildURI())*

N-18 Gas Optimizations - Phase 3

The following opportunities for gas optimizations were identified:

- The `tokenQuota[_token].quota` storage variable is read twice in the `updateQuota` function. Instead, it could be stored as an `oldQuota` stack variable to save gas.
- The `_transferTokens` function first forwards the tokens that are sent to the destination chain or recalled on the source chain and then checks the available quota in

the `QuotaManager`. Assuming that the call more likely fails at the quota check, it would be cheaper to check the available quota first and thereby fail early.

Consider applying the changes above to make the code a little bit more gas efficient.

Update: *Resolved in pull request #17483 at commit 09e5508.*

N-19 Code Quality and Readability Suggestions - Phase 3

The following opportunities to improve the codebase were identified:

- The `DO_TARGET_CALL_REVERTED` error is unused and could be removed.
- The `BTOKEN_INVALID_TO_ADDR` errors [1] [2] [3] are unused and could be removed.
- The `fee` calculation in line 284 of `Bridge.sol` is calculated based on `baseFee`, `maxFee`, and `_message.fee`. The resulting minimal fee is determined through the in-place comparison between `baseFee` and `maxFee`, which harms readability. Instead, consider chaining two `.min()` expressions to simplify the code. For example: `uint256 fee = _message.fee // the fee is at most as provided .min(maxFee) // the capped fee if baseFee >= maxFee .min((maxFee + baseFee) >> 1); // the capped fee if maxFee > baseFee, the profit is halved`
- The `TokenSent` event could include `ctoken.chainid` as canonical tokens are uniquely identified by the combination of their address and their source chain.

Consider applying the above changes to improve the clarity of the codebase.

Update: *Resolved in pull request #17483 at commit 119ffd6 and pull request #17502 at commit 9dd90cc.*

N-20 Typographical Error - Phase 3

A typographical error was identified in `Bridge.sol`. "Owner" has been incorrectly written as "Owenr".

Consider fixing the error to improve the readability of the codebase.

Update: *Resolved in pull request #17303 at commit b63c2c1.*

N-21 Naming Suggestion - Phase 3

In the `ERC20Vault`, the `btokenBlacklist` mapping keeps track of token contracts that have been replaced. Consider renaming the mapping to `btokenBlocklist` or `btokenDenylist`, using a more neutral wording.

Update: *Resolved in pull request #17331 at commit 8495f04.*

N-22 License Incompatibility in `LibBytes` - Phase 3

The `toString` function is used to decode abi encoded `bytes32` or `string` data into a string. This function is taken from another codebase which is licensed under AGPL-3.0.

However, the current codebase is licensed under MIT, which is incompatible with code licensed under AGPL-3.0.

Consider using an MIT-licensed alternative to avoid such an issue (e.g., `BoringERC20`).

Update: *Resolved in pull request #17504 at commit 8b6c137.*

N-23 Missing and Misleading Documentation - Phase 3

Several instances where some documentation was missing or misleading were found:

- When processing a message through the bridge, the message's target and data are checked. While the accompanying comment says "Handle special addresses that don't require actual invocation", the message is simply rejected if the invocation conditions are not satisfied, some of which are not related to the target address. Consider clarifying the comment with the intention of the check.
- The comment about reentrancy for the `DelegateOwner`'s `onMessageInvocation` function is outdated as the function is not intended to be reentered any more and is protected from reentrancy through the `Bridge` contract.
- The comment above the `IBridgedERC20` interface may be outdated as USDC specifically no longer requires an intermediary adapter contract. Other tokens may require one still.
- The `_quota` parameter of the `updateQuota` function is described as "The new daily quota", whereas the quota is measured by the `quotaPeriod`.

- The fee calculation of the `processMessage` function is overall not documented. This makes it more difficult to reason about how the fee is constructed. Consider adding comments to each step of the calculation to explain why it is necessary.
- The documentation around the `IBridgedERC20` interface could add a list of specifications canonical tokens should respect. For example, the code assumes calls to `burn` or `mint` to increase/decrease the balance by exactly the amount given as argument. Tokens with fees on `burn`/`mint`, or taking `type(uint256)` to mean the balance of the sender, are not supported. Similarly, a bridged token should have the same name, symbol, and decimals as its canonical version if these are implemented. This list of specifications could be built up over time to reduce the risk of error with such tokens. An example of supported token specifications can be found here.
- The comment about the amount of gas which should be associated with a transaction calling `processMessage` is misleading. The gas sent with a transaction can in fact be smaller than `(message.gasLimit - GAS_RESERVE) * 64 / 63 + GAS_RESERVE` if the invoked call uses way less gas than specified in the message's gas limit. Additionally, the calculation is not fully accurate as it neglects the message data length that is accounted for in the `getMessageMinGasLimit` function. In addition to correcting the calculation, consider rewording the comment as a recommendation, for example, as follows: "To ensure successful execution, we recommend this transaction's gas limit not to be smaller than: [...]".

Consider addressing the instances identified above to improve the clarity and readability of the codebase.

Update: *Resolved in pull request #17483 and pull request #17546 at commit 7fa3b55.*

Client Reported

CR-01 Incorrect Deletion During Migration in 'ERC20Vault'

The `changeBridgedToken` function can be called by the owner of an `ERC20Vault` contract to migrate a bridged token to another contract.

However, during this migration, a `bridgedToCanonical` mapping slot is deleted for the new token address instead of the old token. The old token is still correctly blacklisted, meaning that it can no longer be bridged, but the mapping should be cleared correctly to make the code clearer and save gas.

Update: *Resolved at commit 42c279f.*

CR-02 Static Calls to 'proveSignalReceived' Revert on State Caching

The `proveSignalReceived` function in the `ISignalService` interface is called in the `_proveSignalReceived` function in the `Bridge` contract using a `staticcall`. However, a static call will revert if the data is cached.

Update: *Resolved at commit dd8725f. The static call was removed, and two functions were made available in* `SignalService`*: a view function without caching, and a separate function with caching.*

Conclusion

The audited codebase contains contracts composing the Taiko rollup, including the rollup contracts themselves, the cross-chain messaging component, the governance system, and the grants contracts.

Overall, we found the codebase to be well-written. Docstrings are present in the code but we would recommend having additional external documentation about any design considerations that could impact users, developers, or block proposers. This includes, for example, the economics of block proposal as well as the differences compared to Ethereum such as unconstrained block times, gas pricing logic, and the difficulty computation. We would also recommend formalizing the roles of the DAO and the security council, and the rules for other chains to connect to Taiko's signal service. Throughout our audit, the Taiko team was responsive and receptive to feedback.