TAIKO

Taiko

Smart Contract Security Assessment

Version: 2.0

February, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Taiko smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Taiko smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Taiko smart contracts.

Overview

Taiko is a tier-based rollup network designed to run on top of the Ethereum Mainnet. As a Base rollup, all proposing and proving is performed on Ethereum, through the use of smart contracts.

The proof system consists of multiple proof tiers, with each tier expected to be more secure than the last. Taiko includes a zero-knowledge prover along with optimistic and SGX proof systems. Additionally, there is a centralised guardian proof system, which is the highest tier.

The core block processing system Taiko has designed is a native bridge to enable the sending of ETH and ERC20 tokens to and from Taiko L2.

Furthermore, this technology can be used recursively, for layer 2 to layer 3 or layer 3 to layer 4 and onwards.

Security Assessment Summary

This review was conducted on the files hosted on the Taiko repository and were assessed at commit 1eea963.

Retesting activities were performed on commit 59c322d.

Specifically, the files in scope are all the Solidity files within the following directories.

- `4844/`
- `bridge/`
- `common/`
- `L1/`
- `L2/`
- `libs/`
- `signal/`
- `thirdparty/`
- `tokenvaults/`

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 27 issues during this assessment. Categorised by their severity:

- Critical: 3 issues.
- High: 5 issues.
- Medium: 7 issues.
- Low: 7 issues.
- Informational: 5 issues.

Note: considering the number of critical and high severity issues identified during this time-boxed engagement, Sigma Prime recommends further security testing on the code base in scope prior to any deployment.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Taiko smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
TKO-01	Fraudulent Messages May Be Sent Through The Bridge	Critical	Resolved
TKO-02	Insufficient Signature Hashing For SGX `addInstances()`	Critical	Resolved
TKO-03	Incorrect Value Assigned To `token` When Recalling Messages	Critical	Resolved
TKO-04	`AssignmentHooks` May Drain Prover TKO When Calling `proposeBlock()`	High	Resolved
TKO-05	SGX `verifyProof()` Allows Front-running To Prevent Proving L2 Blocks	High	Resolved
TKO-06	Prover Payment In ERC20 Tokens Always Reverts	High	Resolved
TKO-07	`toBytes32()` Can Read Out of Bounds	High	Resolved
TKO-08	Migration Of `BridgedERC20` Tokens Is Missing Access Control	High	Resolved
TKO-09	Lack Of Safe Transfer In `ERC20` Prover Payment	Medium	Resolved
TKO-10	`AssignmentHook` Does Not Sign Over `metaHash`	Medium	Resolved
TKO-11	L2 Proposer Can Significantly Bias Difficulty	Medium	Resolved
TKO-12	Pausing Proving For Longer Than `cooldownWindow` May Skip Contesting	Medium	Resolved
TKO-13	Potential Economic Gain When Submitting False Proofs	Medium	Resolved
TKO-14	Reentrancy Vector In `depositEtherToL2()`	Medium	Resolved
TKO-15	`retryMessage()` Can Only Be Called Once	Medium	Resolved
TKO-16	Lack Of Ability To Remove Added SGX Instances	Low	Resolved
TKO-17	Assigned Provers Can Block Proposals	Low	Closed
TKO-18	Blob Reusable Flag Used Incorrectly	Low	Resolved
TKO-19	`getBlockHash()` Will Revert For The First 256 L2 Blocks	Low	Resolved
TKO-20	Proposers Can Bypass Prover Fee and Signature	Low	Closed
TKO-21	Provers Can Reject Time-Consuming Proof Tiers	Low	Closed
TKO-22	EIP-4844 Early Execution Fails Silently	Low	Resolved
TKO-23	Contesting A Correct Proof May Become Economical Viable	Informational	Closed
TKO-24	SGX Verifier Allows Instances To Be Replaced With Themselves	Informational	Closed
TKO-25	Security Council Is Owner Of `TaikoTimelockController` Proxy Allowing Upgrades	Informational	Closed

TKO-26	Miscellaneous General Comments	Informational	Resolved
TKO-27	Gas Optimisations	Informational	Resolved

TKO-01	Fraudulent Messages May Be Sent Through The Bridge		
Asset	`bridge/Bridge.sol & signal/SignalService.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Due to the lack of restrictions on `message.to` and `message.data` when processing a message, it is possible to steal tokens stored in the `ERCxxxxVault` and Ether stored in the `Bridge` by sending a malicious message hash to the `SignalService` contract.

To perform the attack, a message is sent using the `sendMessage()` function from one bridge to another, with the intended receiver being `SignalService.sendSignal()`. The data to send is a hash of an attacker constructed message.

After processing the message, the hash of this malicious message is stored in the `SignalService` contract with a storage slot calculated with `msg.sender = Bridge`.

With the hash of the malicious message now stored in the `SignalService` contract, it is possible to generate a valid proof. The malicious message may then be processed on the bridge of the other chain.

This is a detailed example how an attacker could steal Ether from the bridge:

1. On Chain A:

   - The attacker calls the function `Bridge.sendMessage()` with a message that calls the `SignalService.sendSignal()` function of Chain B. The signal argument used when calling this function should represent a hash of a message that steals Ether from the Chain A bridge. This is an example of a malicious message:

     ```
     Message({
         id: 1337,
         from: ATTACKER,
         srcChainId: ChainAId,
         destChainId: ChainBId,
         owner: ATTACKER,
         to: chainA_bridge_address
         refundTo: ATTACKER,
         value: valueAmountEther,
         fee: 0,
         gasLimit: 0,
         data: "",
         memo: ""
     });
     ```

   - The attacker calculates the hash of this message and put it in inside a message that calls `SignalService.sendSignal()`. This is an example of this message:

```
IBridge.Message memory message = IBridge.Message({
    from: ATTACKER,
    srcChainId: ChainBId,
    destChainId: ChainAId,
    owner: ATTACKER,
    to: SignalService,
    refundTo: Attacker,
    value: 0,
    fee: 0,
    gasLimit: 0,
    data: abi.encodeWithSelector(SignalService.sendSignal.selector, msgHash),
    memo: ""
});
```

2. On Chain B: The attacker calls the function `Bridge.processMessage()`. The signal is now stored and it is possible to generate the proof of it.
3. On Chain A: The attacker calls the function `Bridge.processMessage()` and all the Ether stored in the bridge is transferred to the attacker.

Recommendations

Add restrictions to `message.to`, restricting the smart contracts which can be called by the bridge.

Additionally, restrict the function selector such that it calls a specific function when sending messages through the bridge. For example require the selector matches `onMessageReceived(bytes memory data)`.

Resolution

The issue has been resovled by including additional checks on the `message.to` field, preventing arbitrary addresses from being called.

The `message.to` field is no longer able to be set to the `signalService` in additional to a modifiable blacklist `addressBanned`. The changes can be seen in PR #15577.

TKO-02	Insufficient Signature Hashing For SGX `addInstances()`		
Asset	`L1/verifiers/SgxVerifier.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A bug exists in one of the `addInstances()` functions which allows replacing a current SGX instance with any arbitrary address. The address may be non-SGX and therefore can operate outside of the trusted execution environment.

There are two implementations of `addInstances()`, one restricted to `onlyOwner` and the other allows current SGX instances to add multiple new SGX instances. To prevent side-channel attacks on ECDSA signing, each instance must replace the private key and address each time it generates a signature.

In the code snippet below, it can be seen that `newInstance` is not included in the `signedHash`. There are no other restrictions on the value of `newInstance`. It can therefore be arbitrarily set by `msg.sender` to a non-SGX address.

```
function addInstances(
    uint256 id,
    address newInstance,
    address[] calldata extraInstances,
    bytes calldata signature
)
    external
    returns (uint256[] memory ids)
{
    bytes32 signedHash = keccak256(abi.encode("ADD_INSTANCES", extraInstances));
    address oldInstance = ECDSA.recover(signedHash, signature);
    if (!_isInstanceValid(id, oldInstance)) revert SGX_INVALID_INSTANCE();

    _replaceInstance(id, oldInstance, newInstance); //@audit newInstance is not signed by the old instance

    ids = _addInstances(extraInstances);
}
```

The impact is high as a malicious instance can create an `ADD_INSTANCES` attestation signature for extra instances. The malicious user then sets the `newInstance` parameter to an address where the private key is known. With the malicious address they are able to sign forged SGX proofs which are accepted by `verifyProof()`.

Recommendations

To resolve this issue, include the value of `newInstance` within the `signedHash`.

Furthermore, it is recommended to include a domain separator with `address(this)` and `block.chainid` within `signedHash` to prevent replay of signatures on other contracts or chains. This can be achieved through the use of EIP-712: Typed structured data hashing and signing.

Additionally, domain separation should be added to the function `verifyProof()` to ensure signatures are not re-usable between the functions `addInstances()` and `verifyProof()`.

Resolution

PR #15514 resolves the issue by updating hashing to include the function parameter `newInstance`.

Additionally, domain separation has been included in hashing to prevent replay on other functions or other smart contracts with the same parameters.

TKO-03	Incorrect Value Assigned To `token` When Recalling Messages		
Asset	`tokenvault/ERC20Vault.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

In the function `onMessageRecalled()` in `ERC20Vault.sol`, the calldata intended for `receiveToken()` is decoded to refund the user's tokens. However, during this decoding the `token` variable is assigned the value `user` instead of the token address.

```
(, address token,, uint256 amount) =
  abi.decode(message.data[4:], (CanonicalERC20, address, address, uint256));
```

As seen in the code snippet above, `token` decoded as the second parameter of `message.data`. However, looking at the encoding in the following code snippet, it can be seen that the second parameter, excluding the selector, is `user`.

```
msgData =
abi.encodeWithSelector(this.receiveToken.selector, ctoken, user, to, _balanceChange);
```

Consequently, the contract is unable to refund the user's tokens, as the incorrect `token` address is used. Therefore, any recalled messages from the `ERC20Vault` will revert or result in funds stuck in the contract.

Recommendations

To resolve this issue, ensure `token` is assigned the correct address, found inside the struct `CannonicalERC20.addr`.

Resolution

The recommendation has been implemented in PR #15582.

TKO-04	`AssignmentHooks` May Drain Prover TKO When Calling `proposeBlock()`		
Asset	`L1/libs/LibProposing.sol` & `L1/hooks/AssignmentHook.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

`AssignmentHooks` are used to transfer the `livenessBond` from the assigned prover, by calling `AssignmentHook.onProposedBlock()`. However, there is nothing to prevent calling the same assignment hook repeatedly for a single block proposal. As a result it is possible to drain the entire approval of TKO funds from the assigned prover.

The issue occurs because each `ProverAssignment` contains a signature that is used to confirm the provers acceptance of a block, however nothing is done to invalidate future uses of the same signature and so the signature can repeatedly be used for a single proposal.

Each time the `assignmentHook` is called, the signature is validated and TKO tokens are transferred from the prover to the `TaikoL1` contract. There is an inequality on line [257] in the following snippet, which allows more TKO than required to be transferred from the hooks.

```
235 IERC20 tko = IERC20(resolver.resolve("taiko_token", false));
    uint256 tkoBalance = tko.balanceOf(address(this));
237
    // Run all hooks.
239 // Note that address(this).balance has been updated with msg.value,
    // prior to any code in this function has been executed.
241 for (uint256 i; i < params.hookCalls.length; ++i) {
        // When a hook is called, all ether in this contract will be send to the hook.
243     // If the ether sent to the hook is not used entirely, the hook shall send the Ether
        // back to this contract for the next hook to use.
245     // Proposers shall choose use extra hooks wisely.
        IHook(params.hookCalls[i].hook).onBlockProposed{ value: address(this).balance }(
247         blk, meta, params.hookCalls[i].data
        );
249 }
    // Refund Ether
251 if (address(this).balance != 0) {
        msg.sender.sendEther(address(this).balance);
253 }

255 // Check that after hooks, the Taiko Token balance of this contract
    // have increased by at least config.livenessBond
257 if (tko.balanceOf(address(this)) < tkoBalance + config.livenessBond) { //@audit inequality allow excess transfer
        revert L1_LIVENESS_BOND_NOT_RECEIVED();
259 }
```

The malicious proposer is required to pay the prover's ETH/ERC20 fee each time they called `assignmentHook.onProposedBlock()`. Hence, this attack can be considered a griefing attack since the attacker does not gain funds but will cause the prover to lose more funds than the cost to the attacker. This assumes the liveness bond is significantly more than the prover fee.

Recommendations

The vulnerability should be mitigated in multiple ways:

- Compare `assignmentHook` addresses in `proposeBlock()` and revert if any are the same.
- Make the balance transfer check a strict equality on line [257], the call should only succeed if exactly the `livenessBond` is received by `TaikoL1`.
- Include a nonce in each prover signature and increment after each signature check in `AssignmentHook.onBlockProposal()` to prevent signature reuse.

Resolution

A resolution has been implemented in two parts:

1. Prevent a hook from being called multiple times in `LibProposing`. This can be seen in PR #15492.
2. Enforce a strict equality on the amount of *TKO* tokens sent to `TaikoL1`, stopping the potential for over-payments in a prover assignment. This can be seen in PR #15486.

TKO-05	SGX `verifyProof()` Allows Front-running To Prevent Proving L2 Blocks		
Asset	`L1/verifiers/SgxVerifier.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The function `verifyProof()` is front-runnable by arbitrary users to prevent proving of a block.

There is no access control on the function `verifyProof()`. The intended caller is the smart contract `TaikoL1`, however it is possible for any user to mimic the call from `TaikoL1`. `verifyProof()` is a stateful function which will modify the address of an instance via `_replaceInstance(id, oldInstance, newInstance)`. As a result, if a malicious user calls `verifyProof()` before `TaikoL1` calls `verifyProof()`, the call from `TaikoL1` will revert due to an invalid instance address.

The impact of calling `verifyProof()` is that the prover's call to `TaikoL1.proveBlock()` will fail. While it is difficult to maintain a DoS attack involving front-running for an extended period of time, this may cause delays to block proving, potentially preventing assigned provers or contesters from proving blocks within the assigned windows.

Recommendations

It is recommended to add access control such that `verifyProof()` can only be called via `TaikoL1` or as part of other verifiers in the tiered proving hierarchy.

However, it is important to note that an SGX instance may clear its private key after signing. If this is the case an instance will require an alternate method of updating their key to account for the case where another user has already verified the block. That is because it is not possible to call `proveBlock()` twice at the same block height without contesting or increasing the proof tier.

Resolution

Access control of the function `verifyProof()` has been restricted to `TaikoL1` and higher tier proofs in PR #15514.

TKO-06	Prover Payment In ERC20 Tokens Always Reverts		
Asset	`L1/hooks/AssignmentHook.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

`AssignmentHooks` are used to pay a prover's fee via calling `onProposedBlock()` and transferring the specified asset from the proposer. A proposer can choose to be paid in ETH or any ERC20 token, if a prover chooses payment in an ERC20 token then incorrect code logic causes the transaction to revert.

This revert occurs because line [**116**] specifies the `transferFrom()` sender as `msg.sender` which will always be `TaikoL1.sol` due to the `onlyFromNamed("taiko")` modifier on the `onProposedBlock()` function. The intended sender is the proposer and as a result there will not be a balance of the ERC20 token nor approval for the transfer available at `TaikoL1.sol` leading to the function reverting. This then forces the `TaikoL1.proposeBlock()` parent function to also revert.

```
if (assignment.feeToken == address(0)) {
    if (msg.value < proverFee + input.tip) {
        revert HOOK_ASSIGNMENT_INSUFFICIENT_FEE();
    }

    unchecked {
        refund = msg.value - proverFee - input.tip;
    }

    // Paying Ether
    blk.assignedProver.sendEther(proverFee, MAX_GAS_PAYING_PROVER);
} else {
    if (msg.value < input.tip) {
        revert HOOK_ASSIGNMENT_INSUFFICIENT_FEE();
    }
    unchecked {
        refund = msg.value - input.tip;
    }
    // Paying ERC20 tokens
    IERC20(assignment.feeToken).transferFrom(msg.sender, blk.assignedProver, proverFee);
}
```

Recommendations

Correcting the `msg.sender` on line [**116**] to instead be the proposer will fix this issue.

Resolution

ERC20 payments have been updated to correctly transfer funds from the block proposer to the assigned prover. This can be seen in PR #15486.

TKO-07	`toBytes32()` Can Read Out of Bounds		
Asset	`thirdparty/LibBytesUtils.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

`LibBytesUtils.toBytes32()` takes some `bytes` as input but does not account for inputs with length 0. If an empty bytes object is given as an input to `toBytes32()` it will read past the input and execute an out-of-bounds read, returning the 32 bytes allocated after the input. As such, if an attacker has control over the variable allocated after the input, they could arbitrarily choose the output of `toBytes32()`

```
function toBytes32(bytes memory _bytes) internal pure returns (bytes32) {
    if (_bytes.length < 32) {
        bytes32 ret;
        assembly {
            ret := mload(add(_bytes, 32)) //@audit this may not be initialised by `_bytes`
        }
        return ret;
    }

    return abi.decode(_bytes, (bytes32)); // will truncate if input length >
        // 32 bytes
}
```

The issue is rated as high severity as `toBytes32()` is used in critical parts of the code such as `LibMerkleTrie.verifyInclusionProof()`.

Recommendations

It is recommended to add a check for the case where input-length is zero.

Resolution

The development team have initiated a clean solution where they instead replace the function `toBytes32()` with Solidity native type casting. That is to instead cast `bytes` types directly to `bytes32()`.

The solution can be seen in PR #15565.

TKO-08	Migration Of `BridgedERC20` Tokens Is Missing Access Control		
Asset	`tokenvault/BridgedERC20Base.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

During a migration from an old bridged token (bToken) to a new bToken, the `burn()` function allows anyone to migrate by burning their old bTokens and minting new bTokens. However, no access control is placed on this functionality. As a result, if a user has their bTokens locked in a smart contract (such as a liquidity pool), a malicious actor can call `burn()` on the tokens in the LP. This would result in the user's new bTokens being stuck in the smart contract.

```
function burn(address account, uint256 amount) public nonReentrant whenNotPaused {
    if (migratingAddress != address(0) && !migratingInbound) {
        // Outbound migration
        emit MigratedTo(migratingAddress, account, amount);
        // Ask the new bridged token to mint token for the user.
        IBridgedERC20(migratingAddress).mint(account, amount); //@audit no access control on `account`
    } else if (msg.sender != resolve("erc20_vault", true)) {
        // Bridging to vault
        revert RESOLVER_DENIED();
    }

    _burnToken(account, amount);
}
```

Recommendations

Add access controls such that a user can only `burn()` (migrate) their own bTokens.

One consideration to make with adding access control is that it is no longer possible to fully migrate away from an old bToken. There will most likely always remain some unmigrated tokens. Previously the team could migrate the remaining tokens for inactive users, but this would no longer be possible. This may lead to complications if the new bTokens need to migrated away from at a later date, since a bToken can not have an inbound and an outbound migration simultaneously.

A solution to this second issue could be to allow the owner to migrate other user's tokens.

Resolution

Access control has been added to the function `burn()` such that it may only be called by the address which is migrating tokens. This can be seen in PR #15566.

TKO-09	Lack Of Safe Transfer In `ERC20` Prover Payment		
Asset	`L1/hooks/AssignmentHook.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`AssignmentHooks` are used to pay a prover's fee via calling `onProposedBlock()` and transferring the specified asset from the proposer. A prover can choose to be paid in ETH or any ERC20 token.

Certain ERC20 tokens vary from the ERC20 specifications, which cause the following transfer to the assigned prover to either always fail or to fail silently.

```
IERC20(assignment.feeToken).transferFrom(msg.sender, blk.assignedProver, proverFee);
```

The first issue is that certain tokens such as USDT will not return a `bool`. The `IERC20` interface expects a boolean to be returned and will attempt to decode the return value. If there is no return value, such as in the USDT case the transaction will be reverted. It is therefore not possible to call these tokens.

The second issue is that some ERC20 implementations, such as BAT, do not revert when there is insufficient balance or allowance. Instead, these tokens return `false` and expect the calling contract to handle this case. These tokens are rare and not often used.

Recommendations

It is recommended to make use of the `safeTransferFrom()` function in the OpenZeppelin `SafeERC20` library. This implementation will resolve both issues in checking if a return value exists and if it does, ensuring the value is `true`.

Resolution

OpenZeppelin's safe transfer checks have been added in commit a31b91a and PR 15567.

TKO-10	`AssignmentHook` Does Not Sign Over `metaHash`		
Asset	`L1/hooks/AssignmentHook.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The function `hashAssignment()` does not hash over `assignment.metaHash`, allowing it to be arbitrarily set by the proposer.

```
function hashAssignment(
    ProverAssignment memory assignment,
    address taikoAddress,
    bytes32 blobHash
)
    public
    pure
    returns (bytes32)
{
    return keccak256(
        abi.encode(
            "PROVER_ASSIGNMENT",
            taikoAddress,
            blobHash,
            assignment.feeToken,
            assignment.expiry,
            assignment.maxBlockId,
            assignment.maxProposedIn,
            assignment.tierFees
        )
    );
}
```

The `metaHash` is required to determine the execution of a block. Changes to the `metaHash` will vary the block execution and therefore also the proof generation time. It is an optional field which may be set to zero, if the prover allows any `metaHash`.

A prover signs over `hashAssigment()` to validate they will be the assigned proposer for the block. Without a prover's signature for the `metaHash`, the proposer is able to set this field in the assignment to an arbitrary value, mostly likely the zero hash.

Additionally, the fields `address(this)` and `block.chainId` are not included in the assignment hash. As a result, it would be possible to replay transactions on other deployed contracts or the same contract on alternate chains.

Recommendations

It is recommended to use EIP-712 to hash the structure `ProverAssignment`. EIP-712 will prevent manipulation of any fields and prevent replay on other contracts or chains.

Resolution

The assignment hash has been updated to sign over both `metaHash` and `parentMetaHash` in PRs #15486 and #15498.

TKO-11	L2 Proposer Can Significantly Bias Difficulty		
Asset	`L1/libs/LibProposing.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

A L2 block proposer can repeatedly rearrange transactions before proposing a block to bias difficulty.

The proposer calls the function `proposeBlock()` with either a transaction list or a blob if EIP-4844 has been implemented. In either case the value is set by the proposer which will be recorded as `meta.blobHash`.

```
unchecked {
    meta.difficulty = meta.blobHash ^ bytes32(block.prevrandao * b.numBlocks * block.number);
}
```

The issue occurs in that difficulty is directly based off the value `meta.blobHash` which can be set by the block proposer. Furthermore, the block proposer knows the fields `block.prevrandao`, `b.numBlocks` and `block.number` ahead of time.

By using a guess and check method the proposer is able to heavily influence the value of `meta.difficulty`.

To perform the attack, the proposer will first create a transaction list and set the last transaction as a no-op with calldata `0x01`. The proposers then calculates `meta.difficulty` using the remaining fields. If the difficulty meets some required threshold, the proposer will accept the block. Otherwise, the proposer will increment the no-op transaction calldata to `0x02` and again check if the difficulty meets the required threshold.

The proposer will repeat the process until they have found a transaction list with the required conditions for difficulty.

There are two impacts of having a non-random difficulty. The first is that it is used to calculate `meta.minTier`. Thus, a proposer can determine exactly what proof level to achieve. Second, programs on L2 may be using difficulty as a partial form of randomness, similar to how Taiko uses difficulty to select the proof tier. A non-random difficulty allows for manipulation of smart contracts which use difficulty as a source of randomness.

Recommendations

It is recommended to remove the fields that may be manipulated by the proposer to bias the difficulty.

One option is to use `keccak256(abi.encodePacked(block.prevrandao, b.numBlocks, block.number))`. There are still limitations to this randomness in that it is predictable from the previous epoch boundary when `block.prevrandao` is set if the L1 block is known.

Proposers still have some ability to bias the difficulty, in that they may choose not to propose a block at a certain L1 block height.

Resolution

The calculations for difficulty have been updated as suggested in the recommendation. The implementation can be seen in PR #15568.

TKO-12	Pausing Proving For Longer Than `cooldownWindow` May Skip Contesting		
Asset	`L1/TaikoL1.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

There is a pausing mechanism specifically to prevent proving and verifying blocks. If there are pending state transitions and the proving is paused for longer than `cooldownWindow`, then when unpaused these transitions will be instantly verifiable.

The `pauseProving()` function is an `onlyOwner` function which can prevent calling `proveBlock()` and `verifyBlocks()`. A pending state transition is one which a prover has called `proveBlock()` but not yet `verifyBlock()`. Each pending state must elapse `cooldownWindow` seconds before it is possible to call `verifyBlocks()`. During this pending state is when other users are able to call `proveBlock()` to contest an invalid state transition.

An edge-case exists where a faulty prover or malicious user submits an invalid state transition just before the owner calls `pauseProving()`. The state transition will now be pending and no users are able to call `proveBlock()` to contest this state transition. If the pause period elapses more than `cooldownWindow`, when the unpause occurs it will be possible to call `verifyBlocks()` on the invalid state transition without allowing sufficient time for users to contest.

Furthermore, the owner of `TaikoL1` is the `TaikoTimelockController` governance contract. The governance contract has a timelock which requires a delay before the transaction can be executed. The security council must also adhere to the delay, although the security council is able to skip voting.

Second interesting case occurs when proving is paused for longer than `provingWindow`. If there is a block proposed before or during the pause which does not have state transition then the assigned prover may miss the `provingWindow`.

Any prover who does not submit a proof within `provingWindow` will not be able to prove a block and lose their liveness bond.

Recommendations

Consider storing an `unpauseTime` to represent the last time that proving was unpaused. Require `verifyBlocks()` to check `block.timestamp > max(ts.timestamp, unpauseTime) + cooldownWindow`.

Additionally, modify the access control on `pauseProving()` to include a pausing role which does not need to adhere to a timelock minimum delay.

Resolution

The pausing mechanism has been modified to include an `unpauseTime`. `unpauseTime` is used when calculating assigned prover times and cooldown times in `proverBlock()` and `verifyBlocks()` respectively, as seen in the PR #15585.

TKO-13	Potential Economic Gain When Submitting False Proofs		
Asset	`L1/libs/LibProving.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

An attacker would stand to gain by submitting a false ZK plus SGZ proof to contest a valid optimistic transition.

Say a prover submits a valid transition which has an optimistic proof. The prover pays the liveness bond plus 1,000 TKO as the validity bond.

Now, let's assume a malicious user is able to create a false proof for the `TIER_SGX_AND_PSE_ZKEVM`. If the malicious user submits this false proof by calling `proverBlock()`, they would immediately contest and win against the optimistic proof. By winning a contest the malicious user will be awarded half of the original validity bond.

The cost to the malicious user is 250 TKO as the validity bond for `TIER_SGX_AND_PSE_ZKEVM`. The reward for the malicious user is 500 TKO for contesting and then winning the proof against the optimistic prover. The net gain for the malicious user is 250 TKO.

```
    reward = ts.validityBond / 4;

    // It's important to note that the contester is set to zero
    // for the tier-0 transition. Consequently, we only grant a
    // reward to the contester if it is not a zero-address.
    if (ts.contester != address(0)) {
        tko.transfer(ts.contester, reward + ts.contestBond);
    } else {
        // The prover is also the contester, so the reward is
        // sent to him.
        tko.transfer(msg.sender, reward); //@audit first reward for malicious prover
    }

    // Given that the contester emerges as the winner, the
    // previous blockHash and signalRoot are considered
    // incorrect, and we must replace them with the correct
    // values.
    ts.blockHash = tran.blockHash;
    ts.signalRoot = tran.signalRoot;
}

// Reward this prover.
// In theory, the reward can also be zero for certain tiers if
// their validity bonds are set to zero.
tko.transfer(msg.sender, reward); // @audit second reward for malicious prover
```

The issue is rated as very low likelihood as it requires to create a fraudulent proof at the `TIER_SGX_AND_PSE_ZKEVM` tier, which requires breaking both ZK and SGX in the same manner.

However, if this issue were to occur the malicious user could continually contest with fake proofs for each pending optimistic transition, gaining 250 TKO per fake proof.

Recommendations

To resolve the issue, consider preventing a contester to immediately submit a higher proof. Allow the prover a chance to submit a higher proof themselves, before other users may submit a proof.

Resolution

The tier system has been updated such that higher tiers require a larger bond. The changes have been reflected in PR #15587.

TKO-14	Reentrancy Vector In `depositEtherToL2()`		
Asset	`L1/TaikoL1.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The function `depositEtherToL2()` does not have the `nonReentrant` modifier. It is therefore possible to reenter into `depositEtherToL2()`.

Both `proposeBlock()` and `depositEtherToL2()` read and write to the fields `state.ethDeposits` and `state.slotA.numEthDeposits` and are potential reentrancy vectors.

Additionally, `LibDepositing.depositEtherToL2()` makes a call to the bridge contract which would allow bypassing the `canDepositEthToL2()` check. However, the bridge contract implements a `receive()` function with no code, and so reentrancy is not possible.

The testing team was unable to find, during the allocated time, an exploitable reentrancy vector to negatively impact the contract. Thus, the impact is rated as medium severity.

Recommendations

Add the `nonReentrant` modifier to the function `depositEtherToL2()` in `TaikoL1.sol`.

Resolution

The `nonReentrant` modifier has been added to `depositEtherToL2()` in PR #15569.

TKO-15	retryMessage() Can Only Be Called Once		
Asset	bridge/Bridge.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Even if the argument `isLastAttempt` is set to `false`, the `retryMessage()` function cannot be called again.

According to the natspec comments on [**232-233**], the `isLastAttempt` argument of the function `retryMessage()` specifies if it is the last attempt to retry the message. However, the status of the message is updated to `FAILED` or `DONE` independently of the value of the variable `isLastAttempt`. Then, after calling this function once, it cannot be called again, because the message status check on [**250-252**] would fail.

```
234 function retryMessage(
        Message calldata message,
236     bool isLastAttempt
    )
238     external
        nonReentrant
240     whenNotPaused
        sameChain(message.destChainId)
242 {
        // If the gasLimit is set to 0 or isLastAttempt is true, the caller must
244     // be the message.owner.
        if (message.gasLimit == 0 || isLastAttempt) {
246         if (msg.sender != message.owner) revert B_PERMISSION_DENIED();
        }
248
        bytes32 msgHash = hashMessage(message);
250     if (messageStatus[msgHash] != Status.RETRIABLE) {
            revert B_NON_RETRIABLE();
252     }

254     // Attempt to invoke the messageCall.
        if (_invokeMessageCall(message, msgHash, gasleft())) {
256         // Update the message status to "DONE" on successful invocation.
            _updateMessageStatus(msgHash, Status.DONE);
258     } else {
            // Update the message status to "FAILED"
260         _updateMessageStatus(msgHash, Status.FAILED); //@audit does not check `isLastAttempt'
        }
262 }
```

Recommendations

Change the function logic to match with the specification of the `isLastAttempt` argument.

Resolution

PR #15403 updates `retryMessage()` such that the status will remain as `RETRIABLE` if a message fails and `isLastAttempt = false`.

TKO-16	Lack Of Ability To Remove Added SGX Instances		
Asset	`L1/verifiers/SgxVerifier.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the `SgxVerifier`, there is no way to remove instances after they have been added.

It may be the case that a malicious actor is able to extract the key of an instance. If this were to occur, there is no functionality for the admin to remove a malicious instance once identified.

Each time an instance is created it is assigned an ID. The malicious actor is able to continually sign new SGX proofs with their ID. Additionally, the malicious actor may add additional instances creating new IDs via the function `addInstances()`.

Recommendations

It is not sufficient to simply remove a malicious ID as the attacker could call `addInstances()` to create new IDs. To resolve the issue the admin would first need to pause `addInstances()` and then remove the malicious users IDs and all additional IDs they have created.

Resolution

Functionality to remove instances has been added in PR #15629.

TKO-17	Assigned Provers Can Block Proposals		
Asset	`L1/libs/LibProposing.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Through various means assigned provers can force blocks they are assigned with to revert. While this is disruptive to block sequencing, they do not stand to gain greatly from doing so and so it is unlikely to occur.

Firstly, a prover must have approved the `AssignmentHook` contract to transfer their TKO for use in the liveness bond when the proposer calls `proposeBlock()`, so by frontrunning this call with a reduction in the approval of TKO to the `AssignmentHook` contract, the prover can force a proposal to revert.

Alternatively, a smart contract based prover can make the `isValidSignature()` call on line [**85**] of `AssignmentHook.sol` revert or not return the correct magic bytes. Or, by using a reverting `receive()` function at their own address, force a revert once `sendEther()` on line [**109**] is called.

Finally a prover can propose an additional block such that `meta.id > assignment.maxBlockId` is no longer true on line [**74**] of `AssignmentHook.sol`.

Recommendations

The Taiko team should determine if these methods of preventing block proposal are likely to occur and take measured action related to the risk. The first approval revert could be avoided by making provers maintain a deposit of TKO tokens within the system, though it is to be noted such a system could still be frontrun by a prover withdrawing said tokens unless it was coded with a withdrawal delay window.

Generally this issue is likely to persist in some form and so making proposers aware that this can occur and enacting off-chain solutions such as proposer nodes automatically switching assigned provers after failure may be a more rigorous solution.

Resolution

The development team have opted not to fix this issue. The reasoning is that a proposer is expected to know the prover and establish a partially trusted relationship. The proposer may avoid using this prover in the future if they block proposals.

TKO-18	Blob Reusable Flag Used Incorrectly		
Asset	`L1/libs/LibProposing.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Transaction blobs can be reused, when this happens they have their hash cached which can then be used, so long as the caching occurred within the `blobExpiry` time window.

This functionality appears to be flawed as the `proposeBlock()` call will revert when `isBlobReusable()` returns `true` rather than `false`, which is not the correct behaviour.

```
126 if (params.blobHash != 0) {
        // We try to reuse an old blob
128     if (isBlobReusable(state, config, params.blobHash)) {
            revert L1_BLOB_NOT_REUSEABLE();
130     }
        meta.blobHash = params.blobHash;
132 }
```

Recommendations

Correct the behaviour of line [**128**] by negating the output of `isBlobReusable()`.

Resolution

The recommendation has been implemented in PR #15572.

TKO-19	`getBlockHash()` Will Revert For The First 256 L2 Blocks		
Asset	`L2/TaikoL2.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

`TaikoL2.getBlockHash()` contains arithmetic that will underflow and revert as long as `block.number` is smaller than 256.

```
174 if (blockId >= block.number - 256) return blockhash(blockId);
```

As a result `getBlockHash()` will not work for the first 256 blocks on Taiko's L2. Since this is a view-only function and not used internally the impact is limited.

Recommendations

Consider refactoring the arithmetic to avoid this issue.

Resolution

The arithmetic has been re-organised such that it will add 256 to `blockId` rather than subtract 256 from `block.number`. The changes are reflected in PR #15570.

TKO-20	Proposers Can Bypass Prover Fee and Signature		
Asset	`L1/libs/LibProposing.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

`AssignmentHooks` are used to verify an assigned prover's signature and pay the prover's required fee from the proposer. Because the proposer can specify which assignment hooks to execute, it is possible for them to bypass both prover signature verification and paying the prover for their work.

This occurs because the only check in `proposeBlock()` that the assignment hook has run, is verifying if the TKO liveness bond has been received. A proposer can therefore create their own assignment hook that pays the liveness bond from their own funds and skip the other aspects of the current assignment hook such as verifying the assigned prover's desire to prove the block and payment for the prover's services.

While the liveness bond should be larger than the fee normally paid to the prover in times of volatility, this may not be true and so a malicious proposer may choose to pay the liveness bond themself instead.

Additionally, if the fees or collected metrics are ever altered to include values such as prover efficiency, this may present further issues as this vulnerability allows a proposer to forcibly assign any prover to a block. While a Taiko node running the default prover software may notice they are assigned a block they hadn't approved, it is possible they may lack proving capacity or could be running a custom prover node that is not capable of detecting the extra block.

Recommendations

Multiple approaches could mitigate this vulnerability:

- Hardcoding the first assignment hook address as the pre-existing `assignmentHook.sol` would enforce signature checking and payment to the prover.
- Alternatively, a mapping `blockSignedAndPaid{[}{]}` could be added to the storage of the pre-existing `assignmentHook.sol` that records the hash of the block being signed against, then there could be a check in `LibProposing.proposeBlock()` after the assignment hooks are finished to ensure both the signature and prover payments have been completed.

Resolution

The development team have opted not to fix this issue with the following reasons.

> *Technically can, but there is no economical incentive to do so.* `livenessBond` *shall be much higher so that propser would risk their own asset.*

TKO-21	Provers Can Reject Time-Consuming Proof Tiers		
Asset	`L1/libs/LibProposing.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The proof tier required for a block is calculated during a call to `proposeBlock()`, a lazy prover can detect the tier of a proof and force the transaction suggesting it to revert at no cost to themselves. While this costs the prover nothing, it would affect their reputation and is likely to cause the associated proposer to swap to a different prover.

Some provers are likely to do this either because they lack hardware capable of proving Intel SGX proofs or ZK-proofs, if they are assigned such blocks they would forfeit their liveness bond and so it is preferable to revert on a maximum of 10% of blocks and focus solely on optimistic blocks. In addition, this may be an economic strategy given optimistic proofs require no work from a prover and so can be generated on much lower grade hardware than ZK-proofs, the prover then retains the fee for this proving work without having to pay for computation done.

This revert can be triggered in multiple different ways:

- Setting tier prices provided to the proposer very high for tiers the prover does not wish to prove, then when the proposer attempts to pay the prover fee the transaction will revert as the lack the balance requested.
- Setting tier prices to different values for a custom ERC20 payment token such as a wrapped version of USDC, this token could then implement specific transfer logic that detects the incoming quantity and reverts for quantities relating to proof types the prover does not desire.
- Providing a custom assignment hook that can read the `meta.minTier` it is supplied by `proposeBlock()` and revert if it is not desired, this variant of the attack is more likely to be seen when the proposer is the same entity as the prover.
- If the proposer is also the prover, they can wrap their call to `proposeBlock()` in a smart contract that can determine the `meta.difficulty` and so `meta.minTier` ahead of the call and revert when it is undesirable.

Recommendations

There are several methods to help mitigate this issue, multiple may be chosen to strengthen the system further:

- Proposer nodes can already specify a `BLOCK_PROPOSAL_FEE`: this could be extended to reject fees higher than a certain level or blacklist certain tokens.
- Enabling proof systems similar to Intel SGX available on other processors such as AMD's SEV: while this solution does not solve the attack vector, it widens the number of candidates who can prove blocks, and so reduces the likelihood of this vulnerability occurring.
- Increasing the number of blocks which require SGX or ZK-proofs can make this attack less viable as it results in these rejections occurring more often and proposers are more likely to pick provers who do prove every assigned block, this has the added bonus of making the system more secure.

Resolution

The development team have opted not to fix this issue now but have the following action items:

- Enabling proof systems similar to Intel SGX TEE. Like gramine suppport for AMD-SEV/AWS Nitro.
- Offchain stats and health-checks of provers and prover endpoints.

TKO-22	EIP-4844 Early Execution Fails Silently		
Asset	`4844/Lib4844.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function `evaluatePoint()` will fail silently before the `POINT_EVALUATION_PRECOMPILE_ADDRESS` has been deployed, potentially allowing invalid blob proofs to succeed.

The issue occurs since there will be no bytecode deployed to the address `POINT_EVALUATION_PRECOMPILE_ADDRESS` before the chosen hard fork. As there is no bytecode, a `staticcall()` to this address will return `bool ok = true` as the first parameter. Therefore, calls to `evaluatePoint()` before the hard fork will always succeed irrelevant of the input data.

```
function evaluatePoint(
    bytes32 blobHash,
    uint256 x,
    uint256 y,
    bytes1[48] memory commitment,
    bytes1[48] memory pointProof
)
    internal
    view
{
    if (x >= BLS_MODULUS) revert POINT_X_TOO_LARGE();
    if (y >= BLS_MODULUS) revert POINT_Y_TOO_LARGE();

    (bool ok,) = POINT_EVALUATION_PRECOMPILE_ADDRESS.staticcall(
        abi.encodePacked(blobHash, x, y, commitment, pointProof)
    ); //@audit the call will return `ok = true` if nothing is deployed here
    if (!ok) revert EVAL_FAILED();
}
```

The issue is rated as low severity as the library will only be used after blobs are enabled in the `TaikoL1` contract, which is setting that requires admin access to change.

Recommendations

The silent failure can be prevented in `Lib4844.sol` by ensuring the return of the precompile static call matches the following snippet from EIP-4844.

```
return Bytes(U256(FIELD_ELEMENTS_PER_BLOB).to_be_bytes32() + U256(BLS_MODULUS).to_be_bytes32())
```

Resolution

The recommendation has been implemented in PR #15574, ensuring any calls will revert if point evaluation fails or is not yet deployed.

TKO-23	Contesting A Correct Proof May Become Economical Viable
Asset	`L1/libs/LibVerifying.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

At times of high gas prices on the Ethereum Mainnet, it may be profitable for a proposer or prover to incorrectly contest a block proof in order to not have to verify any blocks in a following call.

When calling `proposeBlock()` or `proveBlock()`, the caller may have to verify between 2-10 blocks: if the `contestingBond` is set too low or gas is too high, then it can become cheaper to sacrifice the `contestingBond` of TKO tokens rather than paying this additional gas; as once a block is contested no further blocks are verified until the dispute is resolved.

Recommendations

The Taiko team should be mindful when setting the `contestingBond` size to ensure it is sufficiently larger than the expected gas cost of verifying the maximum number of blocks per call even if the cost of gas spikes on Ethereum Mainnet.

Resolution

The development team have ensured that the contestation bond will be set more than the gas fees of verifying 10 blocks.

TKO-24	SGX Verifier Allows Instances To Be Replaced With Themselves
Asset	`L1/verifiers/SgxVerifier.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Within the `SgxVerifier` contract, whenever an instance creates a signature, they are required to replace the current address with a new address. This logic is implemented in the function `_replaceInstance()`. There is a lack of checks to ensure `oldInstance` is not the same as `newInstance`.

```
function _replaceInstance(uint256 id, address oldInstance, address newInstance) private {
    instances[id] = Instance(newInstance, uint64(block.timestamp));
    emit InstanceAdded(id, newInstance, oldInstance, block.timestamp);
}
```

The issue is raised as informational as each SGX instance should ensure that the new instance is randomly generated and therefore different to the previous instance within the trusted environment off-chain.

Furthermore, it is possible to replace the instance with a key that has been used previously or is associated with another ID.

Similarly, the functions `addInstances()` allow the same instance address to be added multiple times.

Recommendations

Consider adding a mapping for used addresses and prevent instances from adding or replacing keys if the address has been seen previously.

Resolution

The development team have opted not to fix this issue and provided the following comments.

> *Intended. The rationale behind is: side-channel attacks, this makes it even more difficult to spoof the private key, even tho it needs physical presence on the chip to reverse engineer such situation, so with this solution this is an even added extra security. Also, an assumption is that host tee (raiko) is implemented correctly and not exposing the private key, but storing it in own memory - not in disk, unencrypted.*

TKO-25	Security Council Is Owner Of `TaikoTimelockController` Proxy Allowing Upgrades
Asset	`L1/gov/TaikoTimelockController.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

According to the deploy scripts, the owner of the `TaikoTimelockController` contract will be the security council. The `TaikoTimelockController` is an upgradable proxy, which can be upgraded instantly by the owner. Therefore, it is possible for the security council to upgrade the timelock controller without any governance voting or delays.

The `TaikoTimelockController` could be upgraded, through the proxy, to a contract without any delays and ignores voting from the `TaikoGovernor` contract.

The issue is important since `TaikoTimelockController` is the owner of most other contracts in the system, such as `TaikoL1`. Setting a malicious contract as the controller would allow making arbitrary changes to the protocol and draining all funds in the bridge.

Recommendations

It is recommended to have the `TaikoTimelockController` to own itself. This enforces at least the minimum delay required to elapse before a proxy upgrade.

Resolution

The development team have opted not to fix this issue. Instead, the initial owner will be the security council. After a period of time when the development team is happy with the security maturity of the protocol, they may transfer ownership to the `TaikoTimelockController`.

TKO-26	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Reachable `assert` Statement**

   Related Asset(s): L1/verifiers/PseZkVerifier.sol

   There is an assert statement on line [75] of `PseZkVerifier.sol` which can be reached if an invalid proof is supplied.

   The following assert will be triggered when a prover supplies a `pointProof` and `blobUsed = false`. While it is not the happy path and the transaction should revert for this case, it is desirable to revert with an error message. This will reduce the gas cost for users and provide an explanation as to why.

   ```
   assert(zkProof.pointProof.length == 0);
   ```

   Consider changing the `assert` statement to a `revert` with an error message.

2. **Events Declared But Never Used**

   Related Asset(s): bridge/Bridge.sol

   The events `SignalSent` and `DestChainEnabled` are declared but never used.

   If not needed, remove the unused events.

3. **Optimistic Proofs Denoted By Default Address**

   Related Asset(s): L1/libs/LibProving.sol

   The verifier address for optimistic proofs is denoted by `address(0)` which is also the default returned by values not set in Solidity. This could lead to bypassing proofs if future verifiers are not set up correctly such as not being set in the `AddressManager`, such mistakes might be difficult to notice at first due to the fact the system will misinterpret the missing verifier entry as an optimistic proof and the call to `proveBlock()` would succeed.

   The proving system should first check for an optimistic proof, then if one is not found it can revert should `resolver.resolve()` return the zero address.

4. **Typos**

   Related Asset(s): contracts/*

   Typos were noticed in some file, these should be corrected for clarity:

   - line [198] of `LibProposing.sol` "alghouth" should be "although".
   - line [245] of `LibProposing.sol` "shall choose use extra" should be "should choose to use extra".
   - line [47] of `SgxVerifier.sol` "timstamp" should be "timestamp".
   - line [16] of `L1/hooks/AssignmentHook.sol` "varification" should be "verification".

   Review noted areas and make alterations as seen fit.

5. **`LibRLPReader` Can Only Parse Lists Shorter 32 Elements**

   Related Asset(s): thirdparty/LibRLPReader.sol

   `LibRLPReader.sol` can only parse lists smaller than `MAX_LIST_LENGTH` which is currently set to 32. `LibRLPReader.sol` is used to parse MPT proofs, as such if a proof is longer than 32 nodes it will not be able to be verified. This would require a very large MPT however (impossible considering gas limits), and is thus very unlikely.

   Consider removing or increasing `MAX_LIST_LENGTH`.

6. **Unnecessary Functions in Some Contracts**

   Related Asset(s): signal/SignalService.sol & common/AddressManager.sol

   The `SignalService` contract inherits from `AuthorizableContract` and this latter inherits from `EssentialContract` which also inherits from `OwnerUUPSUpgradable`. So, the contract `SignalService` has the function `pause()` and `unpause()` which is not necessary as the modifiers `whenPaused` and `whenNotPaused` are never used in `SignalService`.

   The contract `AddressManager` inherits from `OwnerUUPSUpgradable`, so `AddressManager` has the `pause()` and `unpause()` function which is not necessary as the modifier `whenNotPaused` is never used in `AddressManager`.

   Change the design of inheritance so that the contracts don't have unnecessary functions. This would also avoid extra bytecodes and save deployment cost.

7. **`sendEther()` With Zero Amount.**

   Related Asset(s): bridge/Bridge.sol & tokenVaults/ERCxxxVault.sol

   In the function `Bridge.processMessage()`, it is possible that the call `refundTo.sendEther(refundAmount)` on L221 is made with `refundAmount == 0`.

   For each `receiveToken()` function of the different ERCxxx vaults, the call `_to.sendEther(msg.value)` is made without checking the `msg.value`, which could be 0.

   To avoid unnecessary external calls and potentially save gas, consider checking that the amount argument used in `sendEther()` is greater than 0.

8. **Duplicate comment**

   Related Asset(s): thirdparty/LibMerkleTrie.sol

   `_getSharedNibbleLength()` has a duplicated NatSpec comment.

   Remove the duplicate comment to improve readability.

9. **Missing Comments**

   Related Asset(s): thirdparty/LibBytesUtils.sol, thirdparty/LibUint512Math.sol & thirdparty/LibRLPReader.sol

   Some functions have functionality or edge cases that are undocumented and may be unexpected.

   - `LibUint512Math.add()` does not revert when an overflow occurs.
   - `LibBytesUtils.toBytes32()` will align to the left if the input is smaller than 32 bytes. For example: input `0xff` will return `0xff00...00`.
   - `LibRLPReader.readBytes32()` will align to the right for an input smaller than 32 bytes.
   - `LibRLPReader.readAddress()` will return `address(0)` on any input with length 1, regardless of the inputs value.

   It is recommended to mention these edge cases in comments to avoid future issues.

10. **High Mainnet Gas May Lead To Degraded Network Performance**

    Related Asset(s): L1/*

    As Taiko settles on the Ethereum Mainnet and does so in a decentralised manner, it is sensitive to gas price fluctuations. ETH deposits, for example, are only processed once a new block is proposed, therefore if Layer 1 gas becomes very expensive, depositing could become impossible as proposers cease to publish `proposeBlock()` transactions due to it not being profitable. This can be further exacerbated by the max deposit queue size of 1024 deposits, meaning that if this cap is reached all further attempts to queue a deposit will revert.

    Likewise, other actions such as proving or verifying blocks may also become too expensive gas-wise for third party actors to run, leading to degraded performance for users transacting on Taiko.

    Inform users of Taiko limitations and best practices during times of high Layer-1 congestion. Determine if it is worthwhile for the Taiko team to run their own proposer & prover set that can maintain the network performance at a loss during high congestion periods.

11. **Comments Link To Broken URLs**

    Related Asset(s): thirdparty/*

    Some files have comments that contain broken links, such as `LibBytesUtils.sol` linking to the optimism repository which no longer works.

    It is recommended to replace these broken links with the updated version, e.g. the optimism-legacy repository.

12. **Confusing Comments Or Variable Names**

    Related Asset(s): L1/gov/TaikoGovernor.sol, L1/tiers/TaikoA6TierProvider.sol & L1/provers/Guardians.sol

    Some variables or comments are misleading and could be made clearer, such as:

    - On line [86] of `TaikoGovernor.sol` the comment refers to "proposer", however as Taiko also uses this term to mean block proposers it is advised to clearly state the votes make a voter become a vote proposer for the governance system not a block proposer.
    - Each tier in `TaikoA6TierProvider.sol` has its own `maxBlocksToVerify` field which is used when verifying after proving a block, this is not used when proposing blocks which use the tier agnostic `config.maxBlocksToVerifyPerProposal`. The tier field should have its name made clearer such as `maxBlocksToVerifyPerProof` to emphasis this.
    - `Guardians.setGuardians()` makes use of arrays called `guardians` and `_guardians` having these arrays, which are the same type, so similarly named makes errors more likely to occur in future alterations. It is advised to make the arrays differ by more than one character such as renaming `_guardians` to `newGuardians`.

    Review noted areas and delete redundant code to save on deployment costs if deemed worthwhile.

13. **Redundant Code**

    Related Asset(s): L1/*

    Some lines of code are redundant and can be deleted. For example

    - [85-87] of `LibVerifying.sol` contain inequalities where we have `x < 0` for an unsigned integer `x` which will never return true. Likewise `config.ethDepositMaxFee >= type(uint96).max` from line [89] is redundant in light of the stronger condition included on line [90] `config.ethDepositMaxFee >= type(uint96).max / config.ethDepositMaxCountPerBlock`.
    - For `TaikoGovernor.sol` and `TaikoTimelockController.sol`, only parent contracts in proxies need the storage `_gap` variable, in these contracts it serves no purpose.
    - In `Guardians.sol` line [46] contains `_minGuardians == 0` which is redundant, the second check covers this because line [42] ensures `guardians.length >= 5` hence `_minGuardians < _guardians.length / 2` is equal or stronger than `_minGuardians < 2`.
    - In `LibProving.sol` on line [170] the condition `proof.tier < meta.minTier` is redundant as the condition that comes directly after it coupled with the check on line [164] ensure this condition is already matched.

Review noted areas and delete redundant code to save on deployment costs if deemed worthwhile.

14. **No Cap For Amount Of ETH Deposits**

    Related Asset(s): L1/libs/LibVerifying.sol

    A stronger condition is suggested for `ethDepositMaxCountPerBlock` in `isConfigValid()` as currently it has no cap and cannot be changed once set without moving to new proxy logic. If the value is set too high it would be possible for all ETH deposits into Taiko to become halted as if the deposit queue grew to such a size where any block proposing reverted. Furthermore, as deposits are processed as part of the block proposal call, it would also freeze any future block proposals for Taiko until the proxy migrated to a new implementation.

    Consider adding an upper bound check to `ethDepositMaxCountPerBlock` in `isConfigValid()`. Alternatively verify the config values have been set correctly after deployment.

15. **Total Supply Of TKO Token Difficult To Determine**

    Related Asset(s): L1/TaikoL1.sol, L1/libs/LibVerifying.sol & L1/libs/LibProving.sol

    In the Taiko system TKO bonds are taken for various actions and stored in the `TaikoL1.sol` contract, if the prover performs an action incorrectly then they can lose their TKO bond, some of which is burnt. This burning occurs simply by leaving the TKO balance unallocated in the `TaikoL1.sol` contract.

    This mechanism makes it difficult for third party organisations to determine the true liquid total supply of TKO as the `TaikoL1.sol` contract both holds tokens awaiting return to provers and those considered burnt and out of the token supply. Furthermore, leaving burnt tokens in an upgradable contract is not advisable for security reasons, it is possible a future vulnerability allows a user to then drain these burnt tokens and this would cause large ecosystem issues once sold.

    Either burnt TKO tokens should be sent to a recognisable burn address such as `address(0)` or a record of the total burnt tokens should be maintained and used to prevent these burnt tokens being transferred again.

16. **Errors Missing From File**

    Related Asset(s): L1/libs/LibProposing.sol

    On line [**38**] it states *"Warning: Any errors defined here must also be defined in TaikoErrors.sol."*; however the errors `L1_TXLIST_OFFSET` and `L1_TXLIST_SIZE` are not included in `TaikoErrors.sol`.

    Ensure all errors included in `LibProposing.sol` are included in `TaikoErrors.sol`.

    Note that there are two unused errors, `L1_TXLIST_OFFSET_SIZE` and `L1_TXLIST_TOO_LARGE` in `TaikoErrors.sol`.

17. **Unused Errors**

    Related Asset(s): L1/TaikoErrors.sol

    The following errors are not used and can be safely removed.

    - `L1_INSUFFICIENT_TOKEN`
    - `L1_INVALID_ADDRESS`
    - `L1_INVALID_AMOUNT`
    - `L1_TXLIST_OFFSET_SIZE`
    - `L1_TXLIST_TOO_LARGE`

18. **Unused Events**

    Related Asset(s): L1/TaikoEvents.sol

    The following events are not used and can be safely removed.

    - `TokenDeposited`
    - `TokenWithdrawn`
    - `TokenCredited`
    - `TokenDebited`

19. **Reachable Overflow**

    Related Asset(s): L1/hooks/AssignmentHook.sol

    It is possible to cause an overflow on line [95] of `AssignmentHook.sol`. The overflow occurs if `input.tip` is set to a value just below 2^{256}. The solidity compiler will have built in overflow checking for this case and thus, when triggered, the transaction will revert.

```
93  uint256 refund;
    if (assignment.feeToken == address(0)) {
95      if (msg.value < proverFee + input.tip) { //@audit proverFee + input.tip can overflow
            revert HOOK_ASSIGNMENT_INSUFFICIENT_FEE();
97      }
```

    Consider setting a maximum bound on `input.tip`.

20. **Inaccuracy In Comment About Required Check**

    Related Asset(s): L1/libs/LibProving.sol

    The comment in the following code snippet is not correct. The check `blk.metaHash != keccak256(abi.encode(meta))` is necessary to ensure the supplied parameters for `proveBlock()` match those generated in `proposeBlock()`. Without this check, a malicious prover could change a range of fields such as `meta.isBlobUsed`.

```
// Check the integrity of the block data. It's worth noting that in
// theory, this check may be skipped, but it's included for added
// caution.
if (blk.blockId != meta.id || blk.metaHash != keccak256(abi.encode(meta))) {
    revert L1_BLOCK_MISMATCH();
}
```

    Remove the sentence about skipping the check.

21. **Provers May Contest Their Own Proof**

    Related Asset(s): L1/libs/LibProving.sol

    It is possible for a prover to contest their own state transition. The malicious user would lose a portion of their bond depending on which root ends up being correct. Thus, it is not an economically viable attack.

    Consider preventing the prover from contesting their own transition.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team have addressed issues where appropriate in PRs #15600 and #15605.

TKO-27	Gas Optimisations
Asset	`/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Some areas of the protocol could be altered to save gas:

- line [**51**] of `Guardians.sol` uses a storage variable's length in a loop, this should be cached in a local variable to prevent storage being accessed on every loop.
- store `_approvals[version][hash]` in memory rather than accessing it twice in a row on [**88-89**] of `Guardians.sol` .

Recommendations

Review noted areas and make alterations as seen fit.

Resolution

Applicable gas optimisations have been added in PR #15576.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `Foundry` framework was used to perform these tests and the output is given below.

```
Running 2 tests for test/LibUint512Math.t.sol:LibUint512MathTest
[PASS] test_add() (gas: 1344)
[PASS] test_mul() (gas: 1682)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 375.85s

Running 30 tests for test/LibRLPReader.t.sol:LibRLPReaderTest
[PASS] test_readAddress() (gas: 9268)
[PASS] test_readAddress_empty() (gas: 3659)
[PASS] test_readAddress_largerThan20() (gas: 3641)
[PASS] test_readAddress_list() (gas: 3668)
[PASS] test_readAddress_smallerThan20() (gas: 3707)
[PASS] test_readBool() (gas: 1180)
[PASS] test_readBool_empty() (gas: 3655)
[PASS] test_readBool_largerThan1() (gas: 3631)
[PASS] test_readBool_not1or0() (gas: 3708)
[PASS] test_readBool_null() (gas: 3787)
[PASS] test_readBytes() (gas: 23378)
[PASS] test_readBytes32() (gas: 3151)
[PASS] test_readBytes32_empty() (gas: 3641)
[PASS] test_readBytes32_largerThan32() (gas: 3682)
[PASS] test_readBytes32_list() (gas: 4032)
[PASS] test_readBytes_empty() (gas: 3696)
[PASS] test_readBytes_list() (gas: 4018)
[PASS] test_readList() (gas: 36889)
[PASS] test_readList_empty() (gas: 3649)
[PASS] test_readList_invalidLength() (gas: 3910)
[PASS] test_readList_nonList() (gas: 4011)
[PASS] test_readList_null() (gas: 3967)
[PASS] test_readRawBytes() (gas: 28125)
[PASS] test_readString() (gas: 5921)
[PASS] test_readString_empty() (gas: 3716)
[PASS] test_readString_list() (gas: 4018)
[PASS] test_readUint256() (gas: 3342)
[PASS] test_readUint256_empty() (gas: 3752)
[PASS] test_readUint256_largerThan32() (gas: 3681)
[PASS] test_readUint256_list() (gas: 4053)
Test result: ok. 30 passed; 0 failed; 0 skipped; finished in 2.25ms

Running 6 tests for test/LibSecureMerkleTrie.t.sol:LibSecureMerkleTrieTest
[PASS] test_get() (gas: 116833)
[PASS] test_verifyInclusionProof_fakeIntermediateNode() (gas: 70427)
[PASS] test_verifyInclusionProof_fakeRoot() (gas: 68366)
[PASS] test_verifyInclusionProof_fakeValue() (gas: 116644)
[PASS] test_verifyInclusionProof_realProof() (gas: 238686)
[PASS] test_verifyInclusionProof_simple() (gas: 116627)
Test result: ok. 6 passed; 0 failed; 0 skipped; finished in 5.43ms

Running 3 tests for test/AddressManager.t.sol:AddressManagerTest
[PASS] test_getAddress() (gas: 19365)
[PASS] test_setAddress() (gas: 44692)
[PASS] test_setAddress_caller_not_owner() (gas: 16502)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 13.21ms

Running 4 tests for test/AuthorizableContract.t.sol:AuthorizableContractTest
[PASS] test_authorize() (gas: 45639)
[PASS] test_authorize_invalid_address() (gas: 16486)
[PASS] test_authorize_invalid_label() (gas: 44570)
[PASS] test_isAuthorizedAs() (gas: 45621)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 14.06ms

Running 3 tests for test/AddressResolver.t.sol:AddressResolverTest
[PASS] test_resolve() (gas: 111633)
```

```
[PASS] test_resolve_return_zero_address() (gas: 24028)
[PASS] test_resolve_revert_zero_address() (gas: 40773)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 14.09ms

Running 5 tests for test/OwnerUUPSUpgradable.t.sol:OwnerUUPSUpgradableTest
[PASS] test__OwnerUUPSUpgradable_init() (gas: 16060)
[PASS] test_pause() (gas: 24637)
[PASS] test_pause_invalid_owner() (gas: 18111)
[PASS] test_unpause_when_unpaused() (gas: 15797)
[PASS] test_upause() (gas: 25962)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 13.35ms

Running 12 tests for test/PseZkVerifier.t.sol:PseZkVerifierTest
[PASS] test_init() (gas: 18194)
[PASS] test_verifyProof() (gas: 45016)
[PASS] test_verifyProof_invalidEncodingZkEvmProof() (gas: 17325)
[PASS] test_verifyProof_invalidInstanceA() (gas: 27119)
[PASS] test_verifyProof_invalidInstanceB() (gas: 27089)
[PASS] test_verifyProof_invalidVerifierId() (gas: 43718)
[PASS] test_verifyProof_invalidZkpLength() (gas: 23266)
[PASS] test_verifyProof_isBlobUsed() (gas: 103794)
[PASS] test_verifyProof_isContesting() (gas: 13979)
[PASS] test_verifyProof_plonkVerifierInvalidLength() (gas: 47534)
[PASS] test_verifyProof_plonkVerifierInvalidRetrun() (gas: 47612)
[PASS] test_verifyProof_plonkVerifierRevert() (gas: 67853)
Test result: ok. 12 passed; 0 failed; 0 skipped; finished in 16.25ms

Running 19 tests for test/Bridge.t.sol:BridgeTest
[PASS] test_init() (gas: 20342)
[PASS] test_isDestChainEnabled() (gas: 31985)
[PASS] test_isMessageSent() (gas: 128385)
[PASS] test_processMessage() (gas: 87096)
[PASS] test_processMessage_failedCall() (gas: 236897)
[PASS] test_processMessage_invoke_target_address() (gas: 146064)
[PASS] test_processMessage_refundToBob() (gas: 96900)
[PASS] test_proveMessageFailed() (gas: 49105)
[PASS] test_proveMessageReceived() (gas: 49220)
[PASS] test_recallMessage_Twice() (gas: 90594)
[PASS] test_recallMessage_for_ether() (gas: 87694)
[PASS] test_recallMessage_wrongSrcChain() (gas: 20117)
[PASS] test_retryMessage_call_fail() (gas: 280282)
[PASS] test_retryMessage_call_succeed() (gas: 231803)
[PASS] test_sendMessage() (gas: 134027)
[PASS] test_sendMessage_invalidDestChainIdCase1() (gas: 45351)
[PASS] test_sendMessage_invalidDestChainIdCase2() (gas: 45600)
[PASS] test_sendMessage_invalidEtherValue() (gas: 45773)
[PASS] test_sendMessage_invalidOwner() (gas: 32171)
Test result: ok. 19 passed; 0 failed; 0 skipped; finished in 19.82ms

Running 3 tests for test/GuardianVerifier.t.sol:GuardianVerifierTest
[PASS] test_init() (gas: 18261)
[PASS] test_verifyProof() (gas: 26922)
[PASS] test_verifyProof_invalidProver() (gas: 27726)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 13.71ms

Running 5 tests for test/LibBytesUtils.t.sol:LibBytesUtilsTest
[PASS] test_slice() (gas: 117295)
[PASS] test_slice2() (gas: 17718)
[PASS] test_toBytes32() (gas: 5388)
[FAIL. Reason: Assertion failed.] test_toBytes32_vuln() (gas: 15080)
[PASS] test_toNibbles() (gas: 31800)
Test result: FAILED. 4 passed; 1 failed; 0 skipped; finished in 1.93ms

Running 3 tests for test/TaikoGovernor.t.sol:TaikoGovernorTest
[PASS] test_execute() (gas: 516897)
[PASS] test_init() (gas: 56311)
[PASS] test_propose() (gas: 298915)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 15.95ms

Running 4 tests for test/PoCs.t.sol:PoCs
```

```
[PASS] testFail_retryMessage_onlyOnce() (gas: 290909)
[PASS] test_addInstances_replaceMaliciousInstance() (gas: 131041)
[PASS] test_processMessage_maliciousMessage() (gas: 158257)
[PASS] test_verifyProof_frontRun() (gas: 86027)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 17.32ms

Running 2 tests for test/LibFixedPointMath.t.sol:LibFixedPointMathTest
[PASS] test_exp() (gas: 6203)
[PASS] test_exp_overflow() (gas: 3129)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 327.45s

Running 14 tests for test/SgxVerifier.t.sol:SgxVerifierTest
[PASS] test_addInstances_owner() (gas: 169786)
[PASS] test_addInstances_ownerEmptyList() (gas: 18883)
[PASS] test_addInstances_ownerZeroAddress() (gas: 67750)
[PASS] test_addInstances_owner_duplicates() (gas: 101432)
[PASS] test_addInstances_replace() (gas: 141373)
[PASS] test_addInstances_replaceInvalidExpiry() (gas: 104930)
[PASS] test_addInstances_replaceInvalidId() (gas: 103132)
[PASS] test_addInstances_replaceInvalidSignature() (gas: 101639)
[PASS] test_init() (gas: 18442)
[PASS] test_verifyProof() (gas: 93557)
[PASS] test_verifyProof_invalidInstance() (gas: 31337)
[PASS] test_verifyProof_invalidProofLength() (gas: 17475)
[PASS] test_verifyProof_invalidSignature() (gas: 26216)
[PASS] test_verifyProof_isContesting() (gas: 13964)
Test result: ok. 14 passed; 0 failed; 0 skipped; finished in 24.03ms

Running 7 tests for test/ERC1155Vault.t.sol:ERC1155VaultTest
[PASS] testFail_onMessageRecalled_a() (gas: 383344)
[PASS] testFail_onMessageRecalled_encode_memoryOverflow() (gas: 376661)
[PASS] testFail_onMessageRecalled_invalidChainID() (gas: 311671)
[PASS] test_init() (gas: 25091)
[PASS] test_onMessageRecalled() (gas: 256221)
[PASS] test_receiveToken() (gas: 256004)
[PASS] test_sendToken() (gas: 277824)
Test result: ok. 7 passed; 0 failed; 0 skipped; finished in 22.55ms

Running 6 tests for test/SignalService.t.sol:SignalServiceTest
[PASS] test_init() (gas: 20061)
[PASS] test_isSignalSent() (gas: 44022)
[PASS] test_isSignalSent_invalid_app() (gas: 13559)
[PASS] test_isSignalSent_invalid_signal() (gas: 13495)
[PASS] test_sendSignal() (gas: 40030)
[PASS] test_sendSignal_invalid_signal() (gas: 15016)
Test result: ok. 6 passed; 0 failed; 0 skipped; finished in 16.38ms

Running 12 tests for test/ERC721Vault.t.sol:ERC721VaultTest
[PASS] testFail_onMessageRecalled_a() (gas: 476187)
[PASS] testFail_onMessageRecalled_encode_memory_overflow() (gas: 446565)
[PASS] testFail_onMessageRecalled_invalidChainID() (gas: 381685)
[PASS] testFuzz_sendToken_revert_if_amount_not_zero(uint256) (runs: 1000, : 32919, : 32919)
[PASS] test_init() (gas: 62154)
[PASS] test_onMessageRecalled() (gas: 329161)
[PASS] test_receiveToken() (gas: 332187)
[PASS] test_sendToken() (gas: 229178)
[PASS] test_sendToken_revert_array_mismatch() (gas: 30484)
[PASS] test_sendToken_revert_invalid_tokenId() (gas: 61303)
[PASS] test_sendToken_revert_not_supported_interface() (gas: 42749)
[PASS] test_sendToken_revert_not_token_owner() (gas: 196002)
Test result: ok. 12 passed; 0 failed; 0 skipped; finished in 193.17ms

Running 7 tests for test/TaikoL1Proposal.t.sol:TaikoL1ProposalTest
[FAIL. Reason: L1_BLOB_NOT_REUSEABLE()] test_blobHashReusable_Vuln() (gas: 721882)
[FAIL. Reason: Proposal did not verify any blocks] test_bypassVerifyBlocks_vuln() (gas: 1320748)
[FAIL. Reason: Prover should have paid bond] test_fakeHook_Vuln() (gas: 929934)
[FAIL. Reason: HOOK_ASSIGNMENT_INSUFFICIENT_FEE()] test_ignoreHigherTierProofsProposal_Vuln() (gas: 879643)
[PASS] test_proposeTaikoBlock() (gas: 2487813)
[FAIL. Reason: ERC20: insufficient allowance] test_proverPaymentInERC20_vuln() (gas: 520926)
[FAIL. Reason: Wrong liveness bond received] test_repeatHook_Vuln() (gas: 674378)
```

```
Test result: FAILED. 1 passed; 6 failed; 0 skipped; finished in 463.94ms

Running 10 tests for test/TaikoL2.t.sol:TaikoL2Test
[PASS] test_1559Configurable() (gas: 58244)
[PASS] test_anchor_calledTwice() (gas: 188763)
[PASS] test_anchor_invalidBasefee() (gas: 26658)
[PASS] test_anchor_invalidParams() (gas: 162356)
[PASS] test_anchor_invalidSender() (gas: 14445)
[PASS] test_anchor_normal() (gas: 294030)
[FAIL. Reason: Arithmetic over/underflow] test_getBlockHash_vuln() (gas: 13396)
[PASS] test_init() (gas: 25012)
[PASS] test_init_invalid() (gas: 2336038)
[PASS] test_signAnchor(bytes32) (runs: 1000, : 21804, : 21821)
Test result: FAILED. 9 passed; 1 failed; 0 skipped; finished in 566.60ms

Running 10 tests for test/ERC20Vault.t.sol:ERC20VaultTest
[PASS] testFail_onMessageRecalled_decode_revert() (gas: 330998)
[PASS] testFail_onMessageRecalled_invalidChainID() (gas: 266820)
[PASS] testFuzz_20Vault_send_erc20_reverts_invalid_amount(uint256) (runs: 1000, : 27089, : 27089)
[PASS] testFuzz_20Vault_send_revert_if_allowance_not_set(uint256) (runs: 1000, : 57484, : 57484)
[PASS] testFuzz_receiveToken_amount(uint256) (runs: 1000, : 251617, : 251799)
[PASS] testFuzz_receiveToken_msgOwner(address) (runs: 1000, : 251905, : 251905)
[PASS] testFuzz_receiveToken_wrongFrom() (gas: 257736)
[PASS] test_init() (gas: 55383)
[PASS] test_onMessageRecalled() (gas: 357451)
[PASS] test_receiveToken() (gas: 251333)
Test result: ok. 10 passed; 0 failed; 0 skipped; finished in 608.14ms

Ran 21 test suites: 159 tests passed, 8 failed, 0 skipped (167 total tests)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].