Taiko

Scope

The code under review can be found within the C4 Taiko repository.

Summary

Findings

ID	Issues	Severity
H-01	Users will never be able to withdraw their claimed airdrop fully in ERC20Airdrop2.sol contract	High
H-02	Arbitrary coinbase address can be supplied by malicious block proposer to use up another proposer's allowance	High
M-01	Incorrect _Essentialinit() function is used in TaikoToken making snapshooter devoid of calling snapshot()	Medium
M-02	Banned addresses can invoke/execute message call through retryMessage()	Medium
L-01	Consider initializing ContextUpgradeable contract by calling __Context_init() in TaikoToken.sol	Low
L-02	Missing address(0) check for USDC in USDCAdapter	Low

ID	Issues	Severity
L-03	__ERC1155Receiver_init() not initialized in ERC1155Vault	Low
L-04	srcToken and srcChainId is not updated on old token after migration through changeBridgedToken()	Low
L-05	MerkleClaimable does not check if claimStart is less than claimEnd	Low
L-06	If amountUnlocked in TimelockTokenPool is less than 1e18, rounding down occurs	Low
L-07	sendSignal() calls can be spammed by attacker to relayer	Low
L-08	Consider calling Ownable_init() function in constructor to prevent frontrunning of initializer functions are non-approved attackers	Low
L-09	Add "Zero if owner will process themself" comment to gasLimit instead of fee	Low
L-10	Bridge integration issues with swapping protocols	Low
L-11	sendMessage() does not check if STATUS is equal to NEW	Low
L-12	Protocol does not refund extra ETH but implements strict check	Low
L-13	If a message is suspended before processMessage() is called, the ERC20 tokens on the source chain and Ether are not refunded.	Low
L-14	User loses all Ether if their address is blacklisted on canonical token	Low
L-15	onMessageInvocation checks in _invokeMessageCall() can be bypassed to call arbitrary function from Bridge contract	Low
L-16	Consider reading return value from snapshot() function	Low
L-17	One off error in block sync threshold check to sync chain data	Low
L-18	Guardian proof that is never fully approved by minGuardians is never deleted	Low
L-19	Consider making the TIMELOCK_ADMIN_ROLE undergo a delay when transferring the admin role	Low
L-20	One-off error when evaluating deposits to process with the ring buffer size	Low

ID	Issues	Severity
L-21	tokenURI() in BridgedERC721 does not conform to EIP-721 specification	Low
L-22	ERC20Vault and ERC721Vault do not work with all valid ERC20 and ERC721 compliant tokens	Low
L-23	Guardians cannot be fully removed from the system which can compromise the integrity of the protocol	Low
R-01	Consider implementing changeBridgedToken() and btokenBlacklist for ERC721Vault and ERC1155Vault	Recommendation
R-02	Instead of passing an empty string for the data parameter in NFT vaults on token transfers, allow users to supply data	Recommendation
R-03	Use named imports to improve readability of the code and avoid polluting the global namespace	Recommendation
N-01	Avoid hardcoding data in BridgedERC1155	Non-Critical
N-02	Missing source()/canonical() function on BridgedERC115 contract	Non-Critical
N-03	Using unchecked arithmetic in for loops is handled by solc compiler 0.8.22 onwards	Non-Critical
N-04	Typo in comment in Bytes.sol	Non-Critical
N-05	Incorrect comment regarding gasLimit in processMessage()	Non-Critical
N-06	Use require instead of assert	Non-Critical
N-07	Incorrect natspec comment for proveMessageReceived()	Non-Critical

Gas Optimizations

ID	Gas Optimizations	Instances
G-01	Pack structs tightly to consume less slots and save gas	2
G-02	Remove block.coinbase != address(0) check in function onBlockProposed()	1
G-03	Use do-while loop instead of for loop to save gas	7

ID	Gas Optimizations	Instances
G-04	Instead of accessing `deposits_[i].amount` consider typecasting `data`	1
G-05	Assigning `meta_.txListByteOffset` to 0 not required due to default value being 0	1
G-06	Cache _newGuardians.length to save gas	1
G-07	Instead of assigning guardians.length as guardianId, use i + 1 to save gas	1
G-08	Consider removing redundant skipFeeCheck() condition	1
G-09	Remove redundant nonReentrant modifier	1
G-10	Consider adding refundAmount > 0 check to save gas on unnecessary sendEther() call	1
G-11	No need to explicitly set success variable to false due to default value being false	1
G-12	Consider using if-else block instead of ternary operators	2
G-13	Modifier checks not required on function _verfiyHopProof()	1
G-14	Cache op.token in function _handleMessage() to save gas	1
G-15	Consider using safeBatchTransferFrom() instead of running a for loop	1
G-16	Add a check on source chain to ensure _op.to is not address(0) or destination vault to save full cross-chain execution gas	1
G-17	Avoid returning unnecessary 64 bytes in LibAddress on excessivelySafeCall()	1
G-18	Remove redundant string memo field in Message struct to save 1 slot	1

Analysis Report

Findings

[H-01] Users will never be able to withdraw their claimed airdrop fully in ERC20Airdrop2.sol contract

Impact

Context: The ERC20Airdrop2.sol contract is for managing Taiko token airdrop for eligible users, but the withdrawal is not immediate and is subject to a withdrawal window.

Users can claim their tokens within claimStart and claimEnd. Once the claim window is over at claimEnd, they can withdraw their tokens between claimEnd and claimEnd + withdrawalWindow. During this withdrawal period, the tokens unlock linearly i.e. the tokens only become fully withdrawable at claimEnd + withdrawalWindow.

Issue: The issue is that once the tokens for a user are fully unlocked, the withdraw() function cannot be called anymore due to the ongoingWithdrawals modifier having a strict `claimEnd + withdrawalWindow < block.timestamp` check in its second condition.

Impact: Although the tokens become fully unlocked when block.timestamp = claimEnd + withdrawalWindow, it is extremely difficult or close to impossible for normal users to time this to get their full allocated claim amount. This means that users are always bound to lose certain amount of their eligible claim amount. This lost amount can be small for users who claim closer to claimEnd + withdrawalWindow and higher for those who partially claimed initially or did not claim at all thinking that they would claim once their tokens are fully unlocked.

Proof of Concept

Coded POC

How to use this POC:

- Add the POC to `test/team/airdrop/ERC20Airdrop2.t.sol`
- Run the POC using `forge test --match-test testAirdropIssue -vvv`
- The POC demonstrates how alice was only able to claim half her tokens out of her total 100 tokens claimable amount.

```
    function testAirdropIssue() public {
      vm.warp(uint64(block.timestamp + 11));

      vm.prank(Alice, Alice);
      airdrop2.claim(Alice, 100, merkleProof);

      // Roll 5 days after
      vm.roll(block.number + 200);
      vm.warp(claimEnd + 5 days);

      airdrop2.withdraw(Alice);

      console.log("Alice balance:", token.balanceOf(Alice));

      // Roll 6 days after
      vm.roll(block.number + 200);
      vm.warp(claimEnd + 11 days);

      vm.expectRevert(ERC20Airdrop2.WITHDRAWALS_NOT_ONGOING.selector);
```

```
        airdrop2.withdraw(Alice);
    }
```

Logs

```
Logs:
 > MockERC20Airdrop @ 0x0000000000000000000000000000000000000000
   proxy      : 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
   impl       : 0x2e234DAe75C793f67A35089C9d99245E1C58470b
   owner      : 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496
   msg.sender : 0x1804c8AB1F12E6bbf3894d4083f33e07309d1f38
   this       : 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496
 Alice balance: 50
```

Tools Used

Manual Review

Recommended Mitigation Steps

In the modifier ongoingWithdrawals(), consider adding a buffer window in the second condition that gives users enough time to claim the fully unlocked tokens.

```
   uint256 constant bufferWindow = X mins/hours/days;

   modifier ongoingWithdrawals() {
       if (claimEnd > block.timestamp || claimEnd + withdrawalWindow <
block.timestamp + bufferWindow) {
           revert WITHDRAWALS_NOT_ONGOING();
       }
       _;
   }
```

[H-02] Arbitrary coinbase address can be supplied by malicious block proposer to use up another proposer's allowance

Impact

During block proposal, a block proposer is required to pay the prover their prover fee in Ether or an ERC20 fee token.

The issue with the current code is that a malicious block proposer could use another proposer's pre-approved allowance to the AssignmentHook.sol contract. This would allow the proposer to propose blocks with nothing at stake while the other block proposer falls victim to the stealing.

The issue has been marked as Medium-severity since although it is the proposer's responsibility to make atomic transactions (i.e. approve + propose), it is the protocol's responsibility to protect its users and attacks among block proposer's themselves in order to prevent a certain group of proposer's who do not delve deeper into learning the code. To better put it, we could think of proposeBlock() on TaikoL1 as a blackbox for the proposers.

Proof of Concept

During block proposal, the functions are called as follows: proposeBlock() on TaikoL1 => proposeBlock on LibProposing, which calls => the onBlockProposed function while looping through the hook.

The issue currently exists in the onBlockProposed function in the Assignment Hook contract. When the proposer pays the fee to the assigned prover, it requires the proposer to approve certain amount of tokens to the contract in order to allow it to transfer (see here).

A block proposer could frontrun the other block proposer's call once the approval is made and supply the coinbase address as the other block proposer's address. This allows the malicious block proposer to use up the tokens that were approved by the other proposer beforehand.

```
File: AssignmentHook.sol
        // Find the prover fee using the minimal tier
        uint256 proverFee = _getProverFee(assignment.tierFees, _meta.minTier);

        // The proposer irrevocably pays a fee to the assigned prover, either in
        // Ether or ERC20 tokens.
        if (assignment.feeToken == address(0)) {
            // Paying Ether
            _blk.assignedProver.sendEther(proverFee, MAX_GAS_PAYING_PROVER);
        } else {
            // Paying ERC20 tokens


            IERC20(assignment.feeToken).safeTransferFrom(
                _meta.coinbase,
                _blk.assignedProver,
                proverFee
            );
```

Tools Used

Manual Review

Recommended Mitigation Steps

The simplest solution to this issue would be to pass the meta.coinbase address as the msg.sender.

[M-01] Incorrect __Essential_init() function is used in TaikoToken making snapshooter devoid of calling snapshot()

Impact

The EssentialContract.sol contract is inherited by the TaikoToken contract. This essential contract contains two __Essential_init() functions, one with an owner parameter only (see here) and the other with owner and address manager parameters (see here).

The issue with the current code is that it uses the __Essential_init() function with the owner parameter only. This would cause the onlyFromOwnerOrNamed("snapshooter") modifier on the snapshot function to not be able to resolve the snapshooter role since the address manager contract was never set during initialization, thus causing a revert.

Due to this:

1. Snapshooter role is denied from taking snapshots.
2. Timely snapshots for certain periods could have failed by the snapshooter since they would have required the owner to jump in by the time the issue was realized.
3. Correct/Intended functionality of the protocol is affected i.e. the snapshooter role assigned to an address cannot ever perform its tasks validly.

Proof of Concept

Here is the whole process:

1. Snapshooter address calls the snapshot() function. The onlyFromOwnerOrNamed("snapshooter") modifier is encountered first.

```
File: TaikoToken.sol
    function snapshot() public onlyFromOwnerOrNamed("snapshooter") {
        _snapshot();
    }
```

2. In the second condition, the modifier calls the resolve() function with the "snapshooter" role as _name in order to check if the caller (msg.sender) is indeed the address approved by the owner.

```
File: EssentialContract.sol
    modifier onlyFromOwnerOrNamed(bytes32 _name) {
        if (msg.sender != owner() && msg.sender != resolve(_name, true))
            revert RESOLVER_DENIED();
        _;
    }
```

3. The resolve() function is called which internally calls the function _resolve(). In the function _resolve(), the condition on Line 69 evaluates to true and we revert. This is because the addressManager address was never set during initialization using the __Essential_init() function with the owner and address manager parameters. Due to this, the snapshooter address is denied from performing it's allocated tasks.

```
File: AddressResolver.sol
64:     function _resolve(
65:         uint64 _chainId,
66:         bytes32 _name,
67:         bool _allowZeroAddress
68:     ) private view returns (address payable addr_) {
69:         if (addressManager == address(0)) revert
RESOLVER_INVALID_MANAGER();
70:
71:         addr_ = payable(
72:             IAddressManager(addressManager).getAddress(_chainId,
_name)
73:         );
74:
75:         if (!_allowZeroAddress && addr_ == address(0)) {
76:             revert RESOLVER_ZERO_ADDR(_chainId, _name);
77:         }
78:     }
```

Tools Used

Manual Review

Recommended Mitigation Steps

In the init() function, consider using the __Essential_init() function with the owner and address manager parameters instead of the __Essential_init() function with the owner parameter. This would allow the snapshooter address to proceed with taking snapshots as expected.

[M-02] Banned addresses can invoke/execute message call through retryMessage()

Impact

When an address is banned, `_message.data` will not be invoked by _invokeMessageCall() on the address.

The issue is that if an address is banned while the message STATUS is RETRIABLE, the function retryMessage() does not stop the call from executing. This breaks the ban restriction on the address and allows `_message.data` to be invoked on it.

Proof of Concept

1. Let's say the bridge_watchdog bans an address using function banAddress().

- On Line 113, it checks if the address is already banned. If not, set it to true on Line 114.

```
File: Bridge.sol
105:     function banAddress(
106:         address _addr,
107:        bool _ban
108:    )
109:        external
110:        onlyFromOwnerOrNamed("bridge_watchdog")
111:        nonReentrant
112:    {
113:        if (addressBanned[_addr] == _ban) revert B_INVALID_STATUS();
114:        addressBanned[_addr] = _ban;
115:        emit AddressBanned(_addr, _ban);
116:    }
```

2. In case of function processMessage(), the banned address is refunded the Ether and its STATUS is set to DONE. This prevents the function _invokeMessageCall() from being called to execute the `_message.data` on the banned address.

```
File: Bridge.sol
301:            // Process message differently based on the target
address
302:            if (
303:                _message.to == address(0) ||
304:                _message.to == address(this) ||
305:                _message.to == signalService ||
306:                addressBanned[_message.to]
307:            ) {
308:                // Handle special addresses that don't require actual
invocation but
309:                // mark message as DONE
310:                refundAmount = _message.value;
311:                _updateMessageStatus(msgHash, Status.DONE);
312:            } else {
```

3. But if the message is already in the RETRIABLE status when the address is banned, the function retryMessage() does not stop _invokeMessageCall() from being called on Line 368 to execute the `_message.data` on the banned address.

```
File: Bridge.sol
350:    function retryMessage(
351:        Message calldata _message,
352:        bool _isLastAttempt
353:    ) external nonReentrant whenNotPaused
sameChain(_message.destChainId) {
354:        // If the gasLimit is set to 0 or isLastAttempt is true, the
caller must
355:        // be the message.destOwner.
356:        if (_message.gasLimit == 0 || _isLastAttempt) {
357:            if (msg.sender != _message.destOwner) revert
B_PERMISSION_DENIED();
358:        }
359:
        bytes32 msgHash = hashMessage(_message);
        if (messageStatus[msgHash] != Status.RETRIABLE) {
            revert B_NON_RETRIABLE();
        }

        // Attempt to invoke the messageCall.

        if (_invokeMessageCall(_message, msgHash, gasleft())) {
            _updateMessageStatus(msgHash, Status.DONE);
        } else if (_isLastAttempt) {
            _updateMessageStatus(msgHash, Status.FAILED);
        }
        emit MessageRetried(msgHash);
    }
```

Tools Used

Manual Review

Recommended Mitigation Steps

Add a check in function retryMessage() to check if the address `_message.to` is banned. If true, refund the Ether and mark the transaction as DONE (or send a signal to retrieve stuck ERC20 tokens on source chain).

[L-01] Consider initializing ContextUpgradeable contract by calling __Context_init() in TaikoToken.sol

Link

ContextUpgradeable is not initialized in TaikoToken.sol contract. This contract is used in ERC20PermitUpgradeable which is used in ERC20VotesUpgradeable. But neither contract initializes this Context contract when the contracts themselves are intialized.

In TaikoToken.sol here, we can see that the below __Context_init() function is not called.

```
File: ContextUpgradeable.sol
    function __Context_init() internal onlyInitializing {
    }

    function __Context_init_unchained() internal onlyInitializing {
    }
```

[L-02] Missing address(0) check for USDC in USDCAdapter

Link

It is important to implement this check in init() functions since they can only be called once.

```
File: USDCAdapter.sol
    function init(address _owner, address _addressManager, IUSDC _usdc) external initializer {
        __Essential_init(_owner, _addressManager);

        usdc = _usdc;
    }
```

[L-03] __ERC1155Receiver_init() not initialized in ERC1155Vault

Link

Consider initializing these functions in an init() function in the ERC1155Vault contract.

```
File: ERC1155ReceiverUpgradeable.sol
    function __ERC1155Receiver_init() internal onlyInitializing {
    }

    function __ERC1155Receiver_init_unchained() internal onlyInitializing {
    }
```

[L-04] srcToken and srcChainId is not updated on old token after migration through changeBridgedToken()

Link

When a token is migrated to another token, the old token still points towards the same srcToken and srcChainId as the new token since they are not updated through changeBridgedToken().

Due to this external dapps integrating and using these values as reference could run into potential issues. Consider clearing them or changing them to some placeholder data representing the src token and chainId but with a prefix.

```
File: BridgedERC20.sol
    function canonical() public view returns (address, uint256) {
        return (srcToken, srcChainId);
    }
```

[L-05] MerkleClaimable does not check if claimStart is less than claimEnd

```
File: MerkleClaimable.sol
    function _setConfig(uint64 _claimStart, uint64 _claimEnd, bytes32 _merkleRoot) private {

        claimStart = _claimStart;
        claimEnd = _claimEnd;
        merkleRoot = _merkleRoot;
    }
```

[L-06] If amountUnlocked in TimelockTokenPool is less than 1e18, rounding down occurs

Link

If amountUnlocked is less than 1e18, round down occurs. This is not a problem since grants will usually be dealing with way higher values and thus higher unlocking. But this would be a problem for team members or advisors getting maybe 10 taiko or less (in case price of taiko is high). So the more frequent the withdrawing there might be chances of losing tokens due to round down.

```
File: TimelockTokenPool.sol
         uint128 _amountUnlocked = amountUnlocked / 1e18; // divide first
```

[L-07] sendSignal() calls can be spammed by attacker to relayer

Link

Since the function is external, an attacker can continuously spam signals to the offchain relayer which is always listening to signals. This would be more cost efficient on Taiko where fees are cheap.

The signals could also be used to mess with the relayer service i.e. by sending a the same signal early by frontrunning a user's bytes32 signal _parameter.

```
File: SignalService.sol
    function sendSignal(bytes32 _signal) external returns (bytes32) {
        return _sendSignal(msg.sender, _signal, _signal);
    }
```

[L-08] Consider calling Ownable_init() function in constructor to prevent frontrunning of initializer functions are non-approved attackers

Link

Most init functions inheriting from EssentialContract do not implement have restrictions on those functions, which allow anyone to initialize them if not done during deployment.

Consider calling the function in the constructor and applying the onlyOwner modifier on the init functions.

```
File: EssentialContract.sol
    constructor() {
        _disableInitializers();
    }
```

[L-09] Add "Zero if owner will process themself" comment to gasLimit instead of fee

In the current code, the preferredExecutor for executing bridged transactions is determined by whether the gasLimit is 0 or not and not the fee.

```
File: IBridge.sol
        // Processing fee for the relayer. Zero if owner will process themself.
        uint256 fee;
        // gasLimit to invoke on the destination chain.
        uint256 gasLimit;
```

[L-10] Bridge integration issues with swapping protocols

Cross-chain swapping could not occur on chains having long invocation delays since deadline of the swap might expire and become outdated. Consider having custom delays for dapps looking to use bridge.

```
File: Bridge.sol
    /// the transactor is not the preferredExecutor who proved this message.
    function getInvocationDelays()
        public
        view
        virtual
        returns (uint256 invocationDelay_, uint256 invocationExtraDelay_)
    {
        if (
            block.chainid == 1 // Ethereum mainnet
        ) {
            // For Taiko mainnet
            // 384 seconds = 6.4 minutes = one ethereum epoch
            return (1 hours, 384 seconds);
        } else if (
            block.chainid == 2 || // Ropsten
            block.chainid == 4 || // Rinkeby
            block.chainid == 5 || // Goerli
            block.chainid == 42 || // Kovan
            block.chainid == 17_000 || // Holesky
            block.chainid == 11_155_111 // Sepolia
        ) {
            // For all Taiko public testnets
            return (30 minutes, 384 seconds);
        } else if (block.chainid >= 32_300 && block.chainid <=
32_400) {
            // For all Taiko internal devnets
            return (5 minutes, 384 seconds);
        } else {
            // This is a Taiko L2 chain where no deleys are applied.
            return (0, 0);
        }
    }
```

[L-11] sendMessage() does not check if STATUS is equal to NEW

Link

Adding a sanity check would be good to avoid being able to call message that is not in the STATUS = NEW state. This would ensure retriable, recalls and failed txns cannot be repeated again.

```
File: Bridge.sol
    function sendMessage(
        Message calldata _message
    )
        external
        payable
        override
        nonReentrant
        whenNotPaused
        returns (bytes32 msgHash_, Message memory message_)
    {
        // Ensure the message owner is not null.
        if (
            _message.srcOwner == address(0) || _message.destOwner == address(0)
        ) {
            revert B_INVALID_USER();
        }

        // Check if the destination chain is enabled.
        (bool destChainEnabled, ) = isDestChainEnabled(_message.destChainId);

        // Verify destination chain and to address.
        if (!destChainEnabled) revert B_INVALID_CHAINID();
        if (_message.destChainId == block.chainid) {
            revert B_INVALID_CHAINID();
        }

        // Ensure the sent value matches the expected amount.

        uint256 expectedAmount = _message.value + _message.fee;
        if (expectedAmount != msg.value) revert B_INVALID_VALUE();

        message_ = _message;

        // Configure message details and send signal to indicate message sending.
        message_.id = nextMessageId++;
        message_.from = msg.sender;
        message_.srcChainId = uint64(block.chainid);

        msgHash_ = hashMessage(message_);

        ISignalService(resolve("signal_service", false)).sendSignal(msgHash_);
        emit MessageSent(msgHash_, message_);
    }
```

[L-12] Protocol does not refund extra ETH but implements strict check

See spec here

The IBridge.sol contract specifies that extra ETH provided when sending a message is refunded back to the user. This currently does not happen since the code implements strict equality check. Using strict equality is better but pointing out the spec described, which would either be followed in the code implemented or the spec should be described properly in the IBridge.sol contract.

```
File: Bridge.sol
        uint256 expectedAmount = _message.value + _message.fee;
        if (expectedAmount != msg.value) revert B_INVALID_VALUE();
```

[L-13] If a message is suspended before processMessage() is called, the ERC20 tokens on the source chain and Ether are not refunded.

If a message is suspended before processMessage() is called, the status of the message remains new and the ERC20 tokens on the source and the Ether is locked as well. If the message will never be unsuspended, consider refunding the tokens to the user.

```
File: Bridge.sol
        if (block.timestamp >= invocationDelay + receivedAt) {
            // If the gas limit is set to zero, only the owner can process the message.
            if (_message.gasLimit == 0 && msg.sender != _message.destOwner) {
                revert B_PERMISSION_DENIED();
            }
```

[L-14] User loses all Ether if their address is blacklisted on canonical token

When recalls are made on the source chain using the function recallMessage(), it calls the onMessageRecalled() function on the ERC20Vault contract. The onMessageRecalled() function transfers the ERC20 tokens back to the user along with any Ether that was supplied.

The issue is with this dual transfer where both ERC20 tokens are Ether are transferred to the user in the same call. If the user is blacklisted on the canonical token, the whole call reverts, causing the Ether to be stuck in the Bridge contract.

To understand this, let's consider a simple example:

1. User bridges ERC20 canonical tokens and Ether from chain A to chain B.
2. The message call on the destination chain B goes into RETRIABLE status if it fails for the first time. (**Note: User can only process after invocation delay**).
3. On multiple retries after a while, the user decides to make a last attempt, on which the call fails and goes into FAILED status.
4. During this time on chain B, the user was blacklisted on the ERC20 canonical token on the source chain.
5. When the failure signal is received by the source chain A from chain B, the user calls recallMessage() on chain A only to find out that although the blacklist is only for the canonical ERC20 token, the Ether is stuck as well.

[L-15] onMessageInvocation checks in _invokeMessageCall() can be bypassed to call arbitrary function from Bridge contract

Link

The if block requires the data to be greater than equal to 4 bytes, equal to the onMessageInvocation selector and last but not the least for the target address to be a contract.

What an attacker could do to bypass this expected spec is to pre-compute an address for the destination chain and pass it in `_message.to`. He can pass gasLimit = 0 from source to only allow him to process the message on the destination.

On the destination chain, the attacker can deploy his pre-computed contract address and call processMessage() with it from the constructor. For a chain (L2s/L3s) with no invocation delays, the proving + executing of the message data would go through in one single call.

When we arrive at the isContract check below on the `_message.to` address, we evaluate to false since the size of the contract during construction is 0. Due to this, the attacker can validly bypass the onMessageInvocation selector that is a requirement/single source of tx origination by the protocol for all transactions occurring from the bridge contract. This breaks a core invariant of the protocol.

```
File: Bridge.sol
        if (
            _message.data.length >= 4 && // msg can be empty
            bytes4(_message.data) !=
            IMessageInvocable.onMessageInvocation.selector &&
            _message.to.isContract()
        ) {
            success_ = false;
        } else {
            (success_, ) = ExcessivelySafeCall.excessivelySafeCall(
                _message.to,
                _gasLimit,
                _message.value,
                64, // return max 64 bytes
                _message.data
            );
        }
```

[L-16] Consider reading return value from snapshot() function

Link

The snapshot() function returns a uint256 snapshotId. These ids if retrieved earlier can make the devs life easier when taking multiple timely snapshots.

```
File: TaikoToken.sol
    function snapshot() public onlyFromOwnerOrNamed("snapshooter") {
        _snapshot();
    }
```

[L-17] One off error in block sync threshold check to sync chain data

The check should be _l1BlockId >= lastSyncedBlock + BLOCK_SYNC_THRESHOLD since threshold is the minimum threshold.

```
File: TaikoL2.sol
        if (_l1BlockId > lastSyncedBlock + BLOCK_SYNC_THRESHOLD) {
            // Store the L1's state root as a signal to the local signal service to
            // allow for multi-hop bridging.
            ISignalService(resolve("signal_service", false)).syncChainData(
                ownerChainId,
                LibSignals.STATE_ROOT,
                _l1BlockId,
                _l1StateRoot
            );
```

Same issue here:

```
File: LibVerifying.sol
        if (_lastVerifiedBlockId > lastSyncedBlock + _config.blockSyncThreshold) {
            signalService.syncChainData(
                 _config.chainId, LibSignals.STATE_ROOT,
_lastVerifiedBlockId, _stateRoot
             );
         }
```

[L-18] Guardian proof that is never fully approved by minGuardians is never deleted

A guardian proof hashs is only deleted if it has been approved by min number of guardians in the approval bits. In case it is not, the approval for the hash remains and is not deleted.

```
File: GuardianProver.sol
         if (approved_) {
             deleteApproval(hash);
             ITaikoL1(resolve("taiko", false)).proveBlock(_meta.id,
abi.encode(_meta, _tran, _proof));
         }
```

[L-19] Consider making the TIMELOCK_ADMIN_ROLE undergo a delay when transferring the admin role

The admin is allowed to skip the delay in operations. But the delay should not be skipped when the role is being transferred.

```
File: TaikoTimelockController.sol
     function getMinDelay() public view override returns (uint256) {
         return hasRole(TIMELOCK_ADMIN_ROLE, msg.sender) ? 0 :
super.getMinDelay();
     }
```

[L-20] One-off error when evaluating deposits to process with the ring buffer size

Link

When calculating the deposits to process, we do not want to overwrite existing slots. This is why the last check/condition is implemented.

The issue with the condition is that it is one-off by the max size the ring bugger allows. Since + 1 is already added, make the check < into <= to work to it's full capacity.

```
File: LibDepositing.sol
         unchecked {

             return
                 _amount >= _config.ethDepositMinAmount &&
                 _amount <= _config.ethDepositMaxAmount &&
                 _state.slotA.numEthDeposits -
                     _state.slotA.nextEthDepositToProcess <
                 _config.ethDepositRingBufferSize - 1;
         }
```

[L-21] tokenURI() in BridgedERC721 does not conform to EIP-721 specification

Impact

The BridgedERC721.sol contract is the bridged representation (on chain B) of a canonical token (on chain A). The issue is that the overriden tokenURI() function in the BridgedERC721 contract does not check whether the `_tokenId` passed as parameter is a valid NFT or not.

Due to this, the ERC721 specification is not conformed to since it requires the tokenURI() function throw/revert if `_tokenId` is not a valid NFT.

```
    /// @notice A distinct Uniform Resource Identifier (URI) for a given
asset.
    /// @dev Throws if `_tokenId` is not a valid NFT. URIs are defined in
RFC
    ///  3986. The URI may point to a JSON file that conforms to the
"ERC721
    ///  Metadata JSON Schema".
    function tokenURI(uint256 _tokenId) external view returns (string);
```

It is essential for tokens created on the Bridge level to conform to EIP specifications in order to not break composability with other smart contracts and consumers such as digital marketplaces, block explorers etc. In our case, the tokenURI() function would build and return an EIP-681 URI for a non-existing tokenId, which could be consumed by external applications that require the Bridged token to be EIP-721 compliant.

Proof of Concept

Below, we can see the difference between the original tokenURI() function that is overriden by the BridgedERC721 contract. On Line 99 in the first snippet, we can see that token existence check is made but when the function was overriden in the second snippet, the check was not applied.

ERC721 tokenURI() function:

```
File: ERC721Upgradeable.sol
098:     function tokenURI(uint256 tokenId) public view virtual override
returns (string memory) {
099:         _requireMinted(tokenId);
100:
101:         string memory baseURI = _baseURI();
102:         return bytes(baseURI).length > 0 ?
string(abi.encodePacked(baseURI, tokenId.toString())) : "";
    }
```

BridgedERC721 tokenURI() overriden function:

```
File: BridgedERC721.sol
    function tokenURI(uint256 _tokenId) public view virtual override
returns (string memory) {
        return string(
            abi.encodePacked(
                LibBridgedToken.buildURI(srcToken, srcChainId),
Strings.toString(_tokenId)
            )
        );
    }
```

Tools Used

Manual Review

Recommended Mitigation Steps

At the start of the function, call the function _requireMinted() to check if the tokenId exists or not.

[L-22] ERC20Vault and ERC721Vault do not work with all valid ERC20 and ERC721 compliant tokens

Impact

According to the README here, all vaults are designed to work with all ERC20, ERC721 and ERC1155 tokens, respectively.

The issue is with the ERC20Vault and ERC721Vault contracts which expect canonical tokens to implement the functions name(), symbol() (in case of both ERC20 and ERC721) and decimals() (in case of ERC20 only).

According to the EIP-20 specification, functions name(), symbol() and decimals() are OPTIONAL and other contracts (in this case the vaults) **MUST NOT** expect these values to be present.

```
OPTIONAL - This method can be used to improve usability, but interfaces
and other contracts MUST NOT expect these values to be present.
```

According to the EIP-721 specification, the metadata extension, which includes functions name() and symbol(), is optional.

```
The metadata extension is OPTIONAL for ERC-721 smart contracts
```

Since both the ERC20Vault and ERC721Vault expect these functions to be present by calling them here and here, the calls would revert. This disallows valid ERC-20 and ERC=721 compliant canonical tokens from being bridged.

The issue does not exist in the ERC1155Vualt since it considers for this situation here by implementing a try-catch block.

Proof of Concept

When the function sendToken() in the ERC20Vault/ERC721Vault is called by the user, it internally calls function _handleMessage(). If the `_token` parameter is a canonical token, we enter the else block on Line 376 for ERC20Vault and Line 204 for ERC721Vault.

In the else block, we can see that the functions name(), symbol() and decimals() are directly being called on the canonical token. Since these are optional, the function will revert and prevent users from bridging valid ERC20 and ERC721 compliant tokens.

Function _handleMessage() in ERC20Vault:

```
File: ERC20Vault.sol
358:     function _handleMessage(
359:         address _user,
360:         address _token,
361:         address _to,
362:         uint256 _amount
363:     )
364:         private
365:         returns (
366:             bytes memory msgData_,
367:             CanonicalERC20 memory ctoken_,
368:             uint256 balanceChange_
369:         )
370:     {
371:         // If it's a bridged token
372:         if (bridgedToCanonical[_token].addr != address(0)) {
373:             ctoken_ = bridgedToCanonical[_token];
374:             IBridgedERC20(_token).burn(msg.sender, _amount);
375:             balanceChange_ = _amount;
376:         } else {
377:             // If it's a canonical token
378:             IERC20Metadata meta = IERC20Metadata(_token);
379:             ctoken_ = CanonicalERC20({
380:                 chainId: uint64(block.chainid),
381:                 addr: _token,
382:                 decimals: meta.decimals(),
383:                 symbol: meta.symbol(),
384:                 name: meta.name()
385:             });
```

Function _handleMessage() in ERC721Vault:

```
File: ERC721Vault.sol
190:     function _handleMessage(
191:         address _user,
192:         BridgeTransferOp memory _op
193:     )
194:         private
195:         returns (bytes memory msgData_, CanonicalNFT memory ctoken_)
196:     {
197:         unchecked {
198:             if (bridgedToCanonical[_op.token].addr != address(0)) {
199:                 ctoken_ = bridgedToCanonical[_op.token];
200:                 for (uint256 i; i < _op.tokenIds.length; ++i) {
201: 
202:                     BridgedERC721(_op.token).burn(_user,
_op.tokenIds[i]);
203:                 }
204:             } else {
205:                 ERC721Upgradeable t = ERC721Upgradeable(_op.token);
206: 
207:                 ctoken_ = CanonicalNFT({
208:                     chainId: uint64(block.chainid),
209:                     addr: _op.token,
210:                     symbol: t.symbol(),
211:                     name: t.name()
212:                 });
```

Unlike the two vaults above, the ERC1155Vault considers this case by implementing a try-catch block on Line 286 for the functions name() and symbol().

Function _handleMessage() in ERC1155Vault:

```
File: ERC1155Vault.sol
260:     function _handleMessage(
261:         address _user,
262:         BridgeTransferOp memory _op
263:     ) private returns (bytes memory msgData_, CanonicalNFT memory
ctoken_) {
264:         unchecked {
265:             // is a btoken, meaning, it does not live on this chain
266:             if (bridgedToCanonical[_op.token].addr != address(0)) {
267:                 ctoken_ = bridgedToCanonical[_op.token];
268:                 for (uint256 i; i < _op.tokenIds.length; ++i) {
269: 
270:                     BridgedERC1155(_op.token).burn(
271:                         _user,
272:                         _op.tokenIds[i],
273:                         _op.amounts[i]
274:                     );
```

```
                }
            } else {
                // is a ctoken token, meaning, it lives on this chain
                ctoken_ = CanonicalNFT({
                    chainId: uint64(block.chainid),
                    addr: _op.token,
                    symbol: "",
                    name: ""
                });
                IERC1155NameAndSymbol t = IERC1155NameAndSymbol(_op.token);

                try t.name() returns (string memory _name) {
                    ctoken_.name = _name;
                } catch {}
                try t.symbol() returns (string memory _symbol) {
                    ctoken_.symbol = _symbol;
                } catch {}
                for (uint256 i; i < _op.tokenIds.length; ++i) {
                    IERC1155(_op.token).safeTransferFrom({
                        from: msg.sender,
                        to: address(this),
                        id: _op.tokenIds[i],
                        amount: _op.amounts[i],
                        data: ""
                    });
                }
            }
        }
```

Tools Used

Manual Review

Recommended Mitigation Steps

Similar to the ERC1155Vault, consider implementing a try-catch block for name(), symbol() and decimals() in ERC20Vault and ERC721Vault.

Note that the BridgedERC20 and BridgedERC721 token contracts would need to be updated as well since they currently validate the name and symbol inputs here and here to be non-zero values. Since according to the EIP, these values can be empty, the calls on destination would always revert. This issue does not exist in BridgedERC1155 since it uses placeholder data here to avoid reverting when a new bridge token is being deployed/created for the canonical token through the vault here.

[L-23] Guardians cannot be fully removed from the system which can compromise the integrity of the protocol

Impact

The end goal of Taiko is to use multiple zkVMs with the guardians removed from the system completely as mentioned here. The issue is that the guardian roles can never be fully removed due to the address(0) check here and 5 minGuardians constraint here. This can compromise the integrity and intended spec of the protocol.

Proof of Concept

When the owner calls the function setGuardians() to remove the guardian roles completely, the following happens:

- On Line 66, we ensure that the _newGuardians array length is above the minimum requirement of 5 guardians.
- On Lines 81-84, the old guardians are deleted.
- On Line 90, the check does not allow the new guardians to be zero.
- Due to this, the owner can never remove the guardians since a minimum requirement of 5 is required and the array cannot contain zero addresses.

```
File: Guardians.sol
059:     function setGuardians(
060:         address[] memory _newGuardians,
061:         uint8 _minGuardians
062:     ) external onlyOwner nonReentrant {
063:         // We need at least MIN_NUM_GUARDIANS and at most 255
guardians (so the approval bits fit in
064:         // a uint256)
065:         if (
066:             _newGuardians.length < MIN_NUM_GUARDIANS ||
067:             _newGuardians.length > type(uint8).max
068:         ) {
069:             revert INVALID_GUARDIAN_SET();
070:         }
071:         // Minimum number of guardians to approve is at least equal
or greater than half the
072:         // guardians (rounded up) and less or equal than the total
number of guardians
073:         if (
074:             _minGuardians < (_newGuardians.length + 1) >> 1 ||
075:             _minGuardians > _newGuardians.length
076:         ) {
077:             revert INVALID_MIN_GUARDIANS();
078:         }
079:
080:         // Delete the current guardians
081:         for (uint256 i; i < guardians.length; ++i) {
082:             delete guardianIds[guardians[i]];
083:         }
084:         delete guardians;
085:
086:         // Set the new guardians
087:
088:         for (uint256 i = 0; i < _newGuardians.length; ++i) {
```

```
            address guardian = _newGuardians[i];
            if (guardian == address(0)) revert INVALID_GUARDIAN();
            // This makes sure there are not duplicate addresses
            if (guardianIds[guardian] != 0) revert INVALID_GUARDIAN_SET();

            // Save and index the guardian
            guardians.push(guardian);
            guardianIds[guardian] = guardians.length;
        }

        // Bump the version so previous approvals get invalidated
        ++version;

        minGuardians = _minGuardians;
        emit GuardiansUpdated(version, _newGuardians);
    }
```

Tools Used

Manual Review

Recommended Mitigation Steps

Consider removing the address(0) check. This would require the owner to sanitize the inputs off-chain.

Another solution would be to implement a separate deletion function that deletes the guardianIds mapping and guardians array and sets them to 0.

[R-01] Consider implementing changeBridgedToken() and btokenBlacklist for ERC721Vault and ERC1155Vault

Link

Both vaults are currently missing these two functions. Implementing them is not required but it would be good as a safety net for high-valued NFT collections in emergency scenarios that could arise.

[R-02] Instead of passing an empty string for the data parameter in NFT vaults on token transfers, allow users to supply data

Allow users to supply the data parameter when transferring tokens from vault to them to ensure any off-chain compatibility/functionality can be built.

```
File: ERC1155Vault.sol
    function _transferTokens(
        CanonicalNFT memory ctoken,
        address to,
        uint256[] memory tokenIds,
        uint256[] memory amounts
    ) private returns (address token) {
```

```
233:         if (ctoken.chainId == block.chainid) {
234:             // Token lives on this chain
235:             token = ctoken.addr;
236:
237:             IERC1155(token).safeBatchTransferFrom(
238:                 address(this),
239:                 to,
240:                 tokenIds,
241:                 amounts,
242:                 ""
243:             );
```

[R-03] Use named imports to improve readability of the code and avoid polluting the global namespace

```
File: LibAddress.sol
4: import "@openzeppelin/contracts/utils/Address.sol";
5: import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
6: import "@openzeppelin/contracts/utils/introspection/IERC165.sol";
7: import "@openzeppelin/contracts/interfaces/IERC1271.sol";
8: import "../thirdparty/nomad-xyz/ExcessivelySafeCall.sol";
```

[N-01] Avoid hardcoding data in BridgedERC1155

Link

Instead of hardcoding the data, place it in a constant variable and assign the variables here for better maintainability.

```
File: BridgedERC1155.sol
53:         LibBridgedToken.validateInputs(_srcToken, _srcChainId, "foo",
"foo");
```

[N-02] Missing source()/canonical() function on BridgedERC115 contract

Link

The BridgedERC1155 contract should implement a similar function to source()/canonical() as done in the other two vaults. This would better for external dapps to retrieve the data much easily.

[N-03] Using unchecked arithmetic in for loops is handled by solc compiler 0.8.22 onwards

```
File: MerkleTrie.sol
205:     function _parseProof(bytes[] memory _proof) private pure returns
(TrieNode[] memory proof_) {
             uint256 length = _proof.length;
             proof_ = new TrieNode[](length);
             for (uint256 i = 0; i < length;) {
                 proof_[i] = TrieNode({ encoded: _proof[i], decoded: RLPReader.readList(_proof[i]) });

                 unchecked {
                     ++i;
                 }
             }
         }
```

[N-04] Typo in comment in Bytes.sol

Use rather instead of rathern.

```
File: Bytes.sol
    /// @notice Slices a byte array with a given starting index up to the end of the original byte
    ///         array. Returns a new array rathern than a pointer to the original.
```

[N-05] Incorrect comment regarding gasLimit in processMessage()

As confirmed with the sponsor, the comment above the gasLimit variable should be inversed i.e. use gasLeft is called by owner, else gasLimit

```
File: Bridge.sol
            } else {
                // Use the specified message gas limit if called by the owner, else
                // use remaining gas

                uint256 gasLimit = msg.sender == _message.destOwner
                    ? gasleft()
                    : _message.gasLimit;
```

[N-06] Use require instead of assert

Use require instead of assert to avoid Panic error, see solidity docs here.

```
File: Bridge.sol
    function _invokeMessageCall(
        Message calldata _message,
        bytes32 _msgHash,
        uint256 _gasLimit
```

```
507:     ) private returns (bool success_) {
508:         if (_gasLimit == 0) revert B_INVALID_GAS_LIMIT();
509:         assert(_message.from != address(this));
```

[N-07] Incorrect natspec comment for proveMessageReceived()

Correct first comment on Line 394 to "msgHash has been received"

```
File: Bridge.sol
394:     /// @notice Checks if a msgHash has failed on its destination chain.
395:     /// @param _message The message.
396:     /// @param _proof The merkle inclusion proof.
397:     /// @return true if the message has failed, false otherwise.
398:     function proveMessageReceived(
399:         Message calldata _message,
400:         bytes calldata _proof
401:     ) public view returns (bool) {
```

Gas Optimizations

[G-01] Pack structs tightly to consume less slots and save gas

Link to instance

The struct currently uses 8 slots. One slot can be reduced by moving the `uint8 blockSyncThreshold` variable on Line 60 after `uint96 livenessBond;` on Line 40. This is because variables from Lines 27 to 40 occupying slot 2 only use up 24 bytes, thus more variable of smaller size can be fit into the slot. The reason the last variable can be moved is to ensure the readability of the code due to the comments by the team grouping them.

```
File: TaikoData.sol
11:     struct Config {
12:         // ---------------------------------------------------------------------------
13:         // Group 1: General configs
14:         // ---------------------------------------------------------------------------
15:         // The chain ID of the network where Taiko contracts are deployed.
16:         uint64 chainId;
17:         // ---------------------------------------------------------------------------
18:         // Group 2: Block level configs
19:         // ---------------------------------------------------------------------------
20:         // The maximum number of proposals allowed in a single block.
21:         uint64 blockMaxProposals;
```

```
        // Size of the block ring buffer, allowing extra space for proposals.
        uint64 blockRingBufferSize;
        // The maximum number of verifications allowed when a block is proposed.
        uint64 maxBlocksToVerifyPerProposal;
        // The maximum gas limit allowed for a block.
        uint32 blockMaxGasLimit;
        // The maximum allowed bytes for the proposed transaction list calldata.
        uint24 blockMaxTxListBytes;
        // The max period in seconds that a blob can be reused for DA.
        uint24 blobExpiry;
        // True if EIP-4844 is enabled for DA
        bool blobAllowedForDA;
        // True if blob can be reused
        bool blobReuseEnabled;
        // ---------------------------------------------------------------------------
        // Group 3: Proof related configs
        // ---------------------------------------------------------------------------
        // The amount of Taiko token as a prover liveness bond
        uint96 livenessBond;
        // ---------------------------------------------------------------------------
        // Group 4: ETH deposit related configs
        // ---------------------------------------------------------------------------
        // The size of the ETH deposit ring buffer.
        uint256 ethDepositRingBufferSize;
        // The minimum number of ETH deposits allowed per block.
        uint64 ethDepositMinCountPerBlock;
        // The maximum number of ETH deposits allowed per block.
        uint64 ethDepositMaxCountPerBlock;
        // The minimum amount of ETH required for a deposit.
        uint96 ethDepositMinAmount;
        // The maximum amount of ETH allowed for a deposit.
        uint96 ethDepositMaxAmount;
        // The gas cost for processing an ETH deposit.
        uint256 ethDepositGas;
        // The maximum fee allowed for an ETH deposit.
        uint256 ethDepositMaxFee;
        // The max number of L2 blocks that can stay unsynced on L1 (a value of zero disables
        // syncing)
        uint8 blockSyncThreshold;
    }
```

Link to instance

The struct BlockParams can be packed by moving the variables from Lines 85-87 to the start of the struct. This would ensure that the first slot now not only contains a 20-byte address but also three variables totalling 7 bytes.

```
File: TaikoData.sol
80:     struct BlockParams {
81:         address assignedProver;
82:         address coinbase;
83:         bytes32 extraData;
84:         bytes32 blobHash;
85:         uint24 txListByteOffset;
86:         uint24 txListByteSize;
87:         bool cacheBlobForReuse;
88:         bytes32 parentMetaHash;
89:         HookCall[] hookCalls;
90:     }
```

[G-02] Remove block.coinbase != address(0) check in function onBlockProposed()

Link to instance

`block.coinbase` is the L1 block builder, which cannot be a zero address. Consider removing the unnecessary check to save gas.

```
File: AssignmentHook.sol
        if (input.tip != 0 && block.coinbase != address(0)) {
            address(block.coinbase).sendEther(input.tip);
        }
```

[G-03] Use do-while loop instead of for loop to save gas

Link to POC

Link to instance

```
File: ERC721Airdrop.sol
        for (uint256 i; i < tokenIds.length; ++i) {
            IERC721(token).safeTransferFrom(vault, user, tokenIds[i]);
        }
```

Link to instance

```
File: Bridge.sol
        for (uint256 i; i < _msgHashes.length; ++i) {
            bytes32 msgHash = _msgHashes[i];
            proofReceipt[msgHash].receivedAt = _timestamp;
            emit MessageSuspended(msgHash, _suspend);
        }
```

Link to instance

```
File: TaikoL2.sol
            for (uint256 i; i < 255 && _blockId >= i + 1; ++i) {
                uint256 j = _blockId - i - 1;
                inputs[j % 255] = blockhash(j);
            }
```

Link to instance

```
File: Guardians.sol
        for (uint256 i; i < guardians.length; ++i) {
            delete guardianIds[guardians[i]];
        }
```

Link to instance

```
File: AssignmentHook.sol
        for (uint256 i; i < _tierFees.length; ++i) {
            if (_tierFees[i].tier == _tierId) return _tierFees[i].fee;
        }
```

Link to instance

```
File: LibDepositing.sol
            for (uint256 i; i < deposits_.length; ) {
                uint256 data = _state.ethDeposits[
                    j % _config.ethDepositRingBufferSize
                ];
                deposits_[i] = TaikoData.EthDeposit({

                    recipient: address(uint160(data >> 96)),
                    amount: uint96(data),
                    id: j
                });


                uint96 _fee = deposits_[i].amount > fee
                    ? fee
116:                        : deposits_[i].amount;
117:
118:                    // Unchecked is safe:
119:                    // - _fee cannot be bigger than deposits_[i].amount
120:                    // - all values are in the same range (uint96) except loop
121:                    // counter, which obviously cannot be bigger than uint95
122:                    // otherwise the function would be gassing out.
123:
124:                    unchecked {
125:                        deposits_[i].amount -= _fee;
126:                        totalFee += _fee;
127:                        ++i;
128:                        ++j;
129:                    }
130:                }
```

Link to instance

```
File: LibProposing.sol
284:                for (uint256 i; i < params.hookCalls.length; ++i) {
285:                    if (uint160(prevHook) >= uint160(params.hookCalls[i].hook)) {
286:                        revert L1_INVALID_HOOK();
287:                    }
288:
289:                    // When a hook is called, all ether in this contract will be send to the hook.
290:                    // If the ether sent to the hook is not used entirely, the hook shall send the Ether
291:                    // back to this contract for the next hook to use.
292:                    // Proposers shall choose use extra hooks wisely.
293:
295:                    IHook(params.hookCalls[i].hook).onBlockProposed{
296:                        value: address(this).balance
297:                    }(blk, meta_, params.hookCalls[i].data);
298:
299:                    prevHook = params.hookCalls[i].hook;
300:                }
```

[G-04] Instead of accessing `deposits_[i].amount` consider typecasting `data`

Link to instance

Instead of accessing deposits_[i].amount on Lines 113 and 115, consider typecasting `data` to uint96 as done on Line 108 to avoid unnecessary memory reads.

```
File: LibDepositing.sol
                deposits_[i] = TaikoData.EthDeposit({

                    recipient: address(uint160(data >> 96)),
                    amount: uint96(data),
                    id: j
                });

                uint96 _fee = deposits_[i].amount > fee
                    ? fee
                    : deposits_[i].amount;
```

[G-05] Assigning `meta_.txListByteOffset` to 0 not required due to default value being 0

Link to instance

Assigning the variable below to 0 is not required since there are not previous assignments made in the else block here and a value was not for it here.

```
File: LibProposing.sol
            meta_.txListByteOffset = 0;
```

[G-06] Cache _newGuardians.length to save gas

Link to instance

Lenght of _newGuardians array is accessed multiple times in the function setGuardians(). Consider caching the value.

```
File: Guardians.sol
        if (
            _newGuardians.length < MIN_NUM_GUARDIANS ||
            _newGuardians.length > type(uint8).max
        ) {
            revert INVALID_GUARDIAN_SET();
        }
        // Minimum number of guardians to approve is at least equal or greater than half the
        // guardians (rounded up) and less or equal than the total number of guardians

        if (
            _minGuardians < (_newGuardians.length + 1) >> 1 ||
            _minGuardians > _newGuardians.length
        ) {
            revert INVALID_MIN_GUARDIANS();
        }
```

[G-07] Instead of assigning guardians.length as guardianId, use i + 1 to save gas

Link to instance

Line 99 accesses length of storage array guardians on every iteration as it grows. Instead of doing that, use i + 1 to replicate the same behaviour since only one guardian is pushed every iteration.

```
File: Guardians.sol
089:         for (uint256 i = 0; i < _newGuardians.length; ++i) {
090:             address guardian = _newGuardians[i];
091:             if (guardian == address(0)) revert INVALID_GUARDIAN();
092:             // This makes sure there are not duplicate addresses
093:             if (guardianIds[guardian] != 0) revert INVALID_GUARDIAN_SET();
094:
095:             // Save and index the guardian
096:
097:
098:             guardians.push(guardian);
099:             guardianIds[guardian] = guardians.length;
100:         }
```

[G-08] Consider removing redundant skipFeeCheck() condition

Link to instance

The function skipFeeCheck() currently returns false as hardcoded. This means that the check is redundant since we always arrive on the second condition in the check below. Consider removing the first condition to save gas.

```
File: TaikoL2.sol
        if (!skipFeeCheck() && block.basefee != basefee) {
            revert L2_BASEFEE_MISMATCH();
        }
```

[G-09] Remove redundant nonReentrant modifier

Link to instance

Reentrancy cannot occur in this function. Additionally, cross-function reentrancy cannot occur as well since function banAddress is restricted to the watchdog.

```
File: Bridge.sol
    function banAddress(
        address _addr,
        bool _ban
    )
        external
        onlyFromOwnerOrNamed("bridge_watchdog")
        nonReentrant
    {
        if (addressBanned[_addr] == _ban) revert B_INVALID_STATUS();
        addressBanned[_addr] = _ban;
        emit AddressBanned(_addr, _ban);
    }
```

[G-10] Consider adding refundAmount > 0 check to save gas on unnecessary sendEther() call

Link to instance

In the else block below, consider adding the check `refundAmount > 0` before Line 374. This will save us gas since in most cases the refundAmount would be 0 for most users. This is because refundAmount is only non-zero and assigned here for special addresses.

```
File: Bridge.sol
368:            if (msg.sender == refundTo) {
369:                refundTo.sendEther(_message.fee + refundAmount);
370:            } else {
371:                // If sender is another address, reward it and refund
the rest
372:                msg.sender.sendEther(_message.fee);
373:
374:                refundTo.sendEther(refundAmount);
375:            }
```

[G-11] No need to explicitly set success variable to false due to default value being false

Link to instance

In function *invokeMessageCall(), the if block sets success* to false in the if block if the conditions are met. This is not required since default value of bool is false.

```
File: Bridge.sol
            success_ = false;
```

[G-12] Consider using if-else block instead of ternary operators

Link to instance

```
File: LibMath.sol
    function min(uint256 _a, uint256 _b) internal pure returns (uint256) {
        return _a > _b ? _b : _a;
    }
```

[G-13] Modifier checks not required on function _verfiyHopProof()

Link to instance

The three modifier checks are not required below since the function is called from proveSignalReceived() here, which already implements the checks.

```
File: SignalService.sol
    function _verifyHopProof(
        uint64 _chainId,
        address _app,
        bytes32 _signal,
        bytes32 _value,
        HopProof memory _hop,
        address _signalService
    )
        internal
        virtual
        validSender(_app)
        nonZeroValue(_signal)
        nonZeroValue(_value)
        returns (bytes32)
```

[G-14] Cache op.token in function _handleMessage() to save gas

Link to instance

`op.token` is accessed multiple times in the function below. Consider caching it to save gas.

```
File: ERC1155Vault.sol
            if (bridgedToCanonical[_op.token].addr != address(0)) {
                ctoken_ = bridgedToCanonical[_op.token];
                for (uint256 i; i < _op.tokenIds.length; ++i) {

                    BridgedERC1155(_op.token).burn(
                        _user,
                        _op.tokenIds[i],
                        _op.amounts[i]
                    );
                }
            } else {
                // is a ctoken token, meaning, it lives on this chain
```

```
278:                    ctoken_ = CanonicalNFT({
279:                        chainId: uint64(block.chainid),
280:                        addr: _op.token,
281:                        symbol: "",
282:                        name: ""
283:                    });
```

[G-15] Consider using safeBatchTransferFrom() instead of running a for loop

Link to instance

ERC1155 supports batch transfers. Instead of transferring each tokenId one-by-one, consider batch transferring them to save gas.

```
File: ERC1155Vault.sol
293:                    for (uint256 i; i < _op.tokenIds.length; ++i) {
294:                        IERC1155(_op.token).safeTransferFrom({
295:                            from: msg.sender,
296:                            to: address(this),
297:                            id: _op.tokenIds[i],
298:                            amount: _op.amounts[i],
299:                            data: ""
300:                        });
301:                    }
```

[G-16] Add a check on source chain to ensure _op.to is not address(0) or destination vault to save full cross-chain execution gas

Link to instance

On Line 284, if the `to` address is address(0) or the vault's address, we revert. This causes the user to recall on the source chain to release their associated assets.

A good way to save the execution gas for the whole cross-chain call is to verify the `_op.to` address on the source chain itself in _handleMessage() function. This would prevent both the users and relayers from spending unnecessary gas for the whole call from chain A to chain B and for the user on recall on chain A.

```
File: ERC20Vault.sol
269:     function onMessageInvocation(
270:         bytes calldata _data
271:     ) external payable nonReentrant whenNotPaused {
272:         (
273:             CanonicalERC20 memory ctoken,
274:             address from,
275:             address to,
276:             uint256 amount
277:         ) = abi.decode(_data, (CanonicalERC20, address, address,
uint256));
```

```

        // `onlyFromBridge` checked in checkProcessMessageContext
        IBridge.Context memory ctx = checkProcessMessageContext();

        // Don't allow sending to disallowed addresses.
        // Don't send the tokens back to `from` because `from` is on the source chain.
        if (to == address(0) || to == address(this)) revert VAULT_INVALID_TO();

        // Transfer the ETH and the tokens to the `to` address
        address token = _transferTokens(ctoken, to, amount);
        to.sendEther(msg.value);

        emit TokenReceived({
            msgHash: ctx.msgHash,
            from: from,
            to: to,
            srcChainId: ctx.srcChainId,
            ctoken: ctoken.addr,
            token: token,
            amount: amount
        });
    }
```

[G-17] Avoid returning unnecessary 64 bytes in LibAddress on excessivelySafeCall()

Function sendEther() is one of the most frequently used functions in the bridging functionality. Returning 64 bytes on Line 32 is not required since the return value is not used anywhere. Removing it will save gas since less bytes would be copied to memory.

```
File: LibAddress.sol
22:     function sendEther(address _to, uint256 _amount, uint256 _gasLimit) internal {
23:         // Check for zero-address transactions
24:         if (_to == address(0)) revert ETH_TRANSFER_FAILED();
25:
26:         // Attempt to send Ether to the recipient address
27:
28:         (bool success,) = ExcessivelySafeCall.excessivelySafeCall(
29:             _to,
30:             _gasLimit,
31:             _amount,
32:             64, // return max 64 bytes
33:             ""
34:         );
```

[G-18] Remove redundant string memo field in Message struct to save 1 slot

The string memo field in the struct is not consumed anywhere on the destination bridge contract i.e. neither the bridge itself nor any external application/vault. Removing it will save 1 slot in the struct, thus reducing gas fees on txs.

```
File: IBridge.sol
    struct Message {
        // Message ID whose value is automatically assigned.
        uint128 id;
        // The address, EOA or contract, that interacts with this bridge.
        // The value is automatically assigned.
        address from;
        // Source chain ID whose value is automatically assigned.
        uint64 srcChainId;
        // Destination chain ID where the `to` address lives.
        uint64 destChainId;
        // The owner of the message on the source chain.
        address srcOwner;
        // The owner of the message on the destination chain.
        address destOwner;
        // The destination address on the destination chain.
        address to;
        // Alternate address to send any refund on the destination chain.
        // If blank, defaults to destOwner.
        address refundTo;
        // value to invoke on the destination chain.
        uint256 value;
        // Processing fee for the relayer. Zero if owner will process themself.
        uint256 fee;
        // gasLimit to invoke on the destination chain.
        uint256 gasLimit;
        // callData to invoke on the destination chain.
        bytes data;
        // Optional memo.
        string memo; //@audit Gas/NC - redundant memo
    }
```

Analysis Report

Approach taken in evaluating the codebase

Time spent on this audit: 21 days (Full duration of the contest)

Day 1

- Consuming resources provide in the README
- Understanding and noting down the logical scope

Day 2-7

- Reviewing base contracts (least inherited)
- Adding inline bookmarks for notes
- Understanding RLP encoding, EIP-4844,

Day 8-12

- Reviewing core libs such as LibDepositing, Proposing, Verifying, Proving
- Adding inline bookmarks for problems in libs
- Gas optimizations for libs

Day 13-14

- Other L1, L2 contracts for taiko

Day 15-19

- Reviewing bridge contracts, timelocktokenpool, airdrop contracts

Day 19-21

- Writing reports

Architecture recommendations

What's unique?

1. Allowing custom processing - Allowing users to take over the processing aspect gives them control over how and when they want to process their messages.
2. On L2s and L2s that have no invocation delays, the team has created an MEV market from the bridge itself since the processing fees are rewarded to the fastest processor.
3. Use to transient storage - The team has used TSTORE and TLOAD in two places. One is to enable cheaper reentrancy locks and the second is to provide external applications with context. Having context is important since it allows anyone e.g. the vaults to verify whether the source sending the transactions is valid.

What's using existing patterns and how this codebase compare to others I'm familiar with

- Comparing Taiko to Starknet, the taiko model is much superior since it takes Ethereum's security and uses it to provide cheaper fees to user. The risk associated with Taiko is lesser as well even though it is a type-1 zkevm. This is because Taiko uses guardians while Starknet implements escape hatches, which are more centralized.

Centralization risks

Actors Involved and their roles

1. The biggest trust assumption in the contract is the owner role handling all the Address manager contracts. This role can pause the contracts at anytime.
2. The second trust assumption is the guardians multisig. Currently, the guardians are trusted and will be removed over time. But since they are the highest tier, the centralization risk in the proving system exists.

3. ANother role is the bridge watchdog. This role can ban and suspend any messages at will. It is the most important risk of the bridge contracts.
4. The snapshooter role has some risks associated since it takes snapshots on the TaikoToken.

There are more roles in the codebase but these are the foremost and most central to the protocol.

Resources used to gain deeper context on the codebase

1. Based Rollups and decentralized sequencing

- Understanding the Concept of Based Rollups
- Based Rollup FAQ
- X space - Part 1
- X space - Part 2

2. Based Contestable Rollup (BCR)

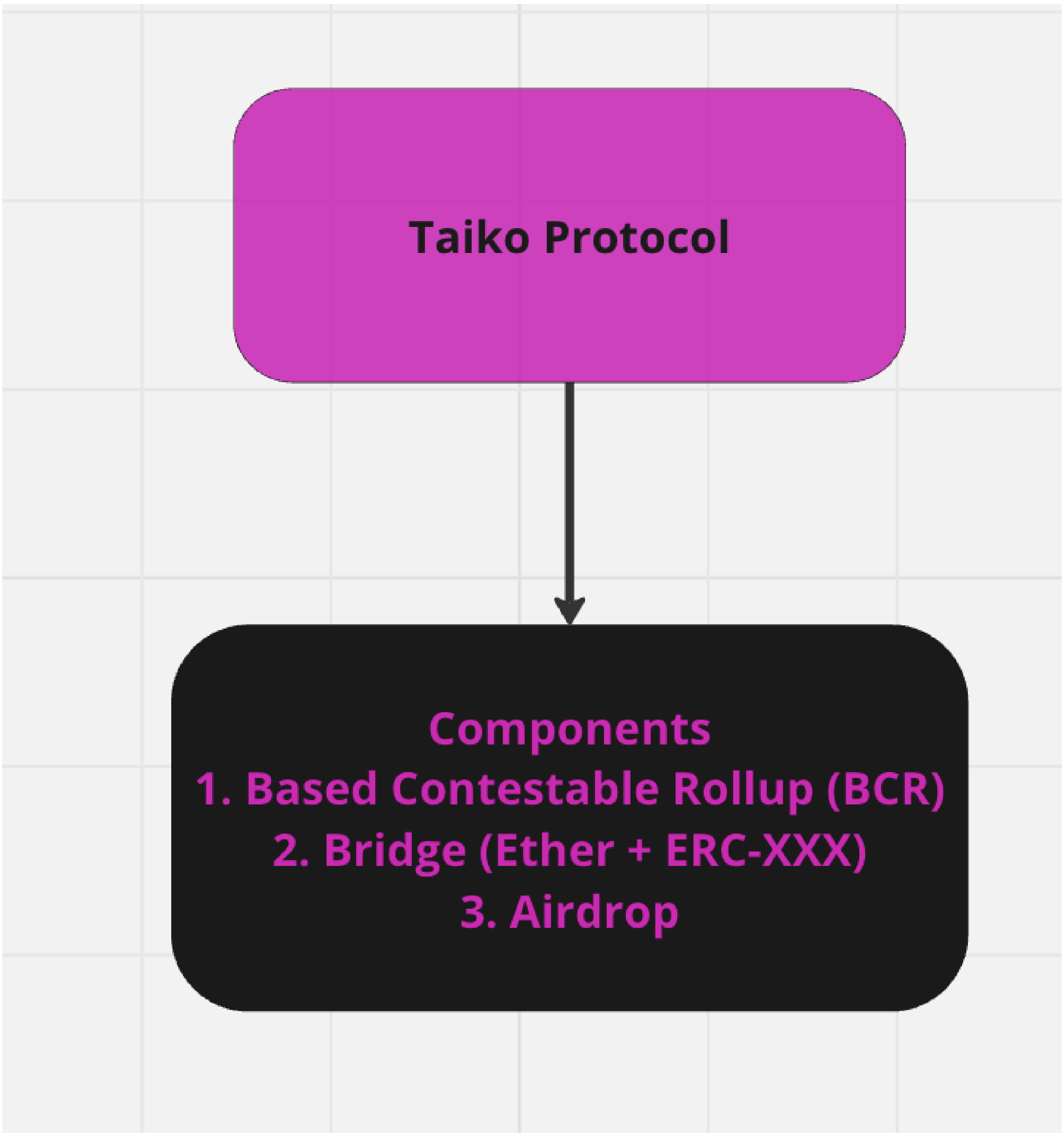
- Understanding the concept of Based Contestable Rollup
- Based Contestable Rollup 101

3. Protocol Documentation

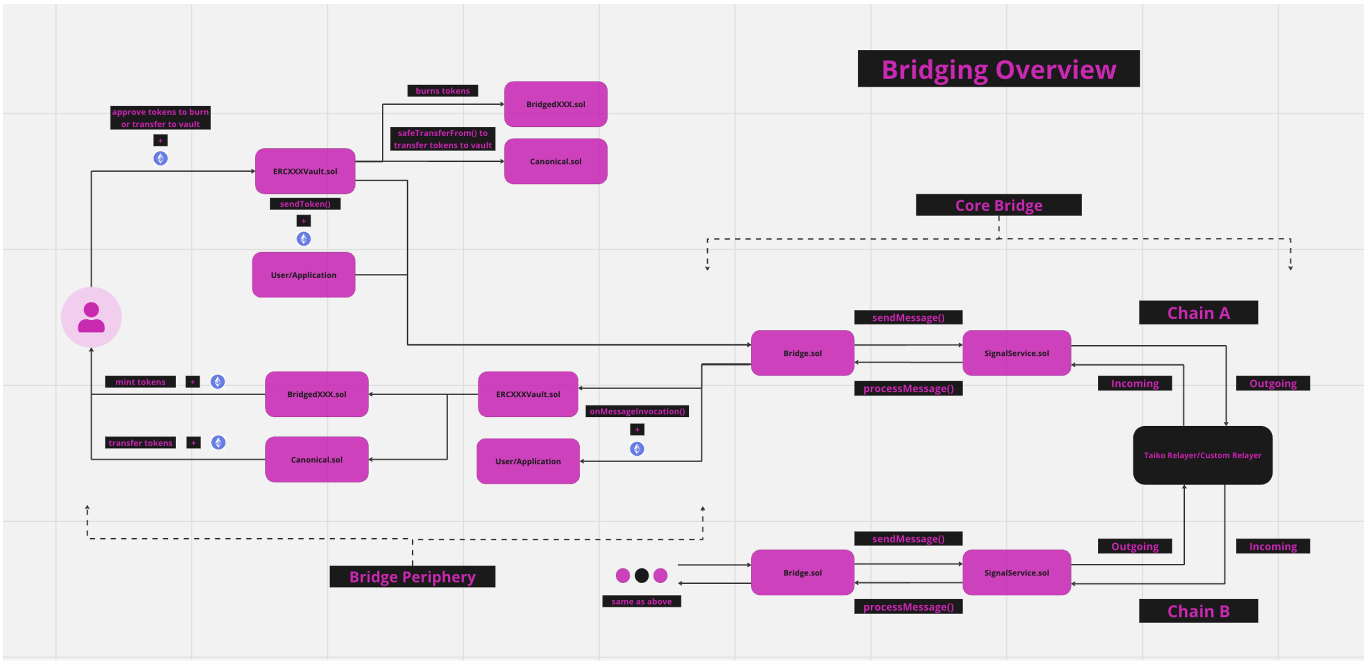
- Website Docs
- Markdown concept-specific docs

Mechanism Review

Protocol Goals


Taiko Protocol
Components
1. Based Contestable Rollup (BCR)
2. Bridge (Ether + ERC-XXX)
3. Airdrop

High Level System Overview



Understanding Taiko BCR

1. Taiko has no sequencer - block builders/sequencing is decided by Ethereum validators. Permissionless block proposing, building and proving of taiko blocks. Since it uses what Ethereum has to provide (the market of proposers/builders) to its advantage and is not reinventing the wheel, Taiko is "based".

2. Why do we need contestable rollups?

Although ZK is the future, it does introduce complexity in the implementation of the system and thus the chances of bugs. There might be a small % of proofs that are invalid but can still be verified onchain. As a protocol designer, Taiko has to handle these small odds. No validity proof can be trusted without battle testing in short. A malicious block proposer (since proposing blocks is permissionless) can bundle a lot of transactions in a block and create an invalid proof, which would not be handleable by the MerkleTrie verification system. This is why contestable rollups are introduced to prevent these kind of situations/attacks.

Due to this since not all blocks are ZK-provable (block gas limit <> ZK constraint limit), Taiko introduced the guardian role to override invalid proofs.

Another reason why this tier-based contestable rollup design is used is in case, app chains, other layer 2s and layer3s want to use different proof systems other than ZK (so opt out of ZK proofs and maybe opt into SGX proof a.k.a Optimistic proofs).

3. How the guardian will be chosen

DAO => Security Council (owner of all smart contracts) => security council is a multisig with a lot of actors not only from taiko but also the community (like token holders) => security council will decide who will be the guardians => guardian is the multisig smart contract onchain => each guardian will have to run their own full node => each guardian works independently and they do not need to reach consensus offchain => they independently correct the wrong onchain => if the aggregated approval onchain is greater than let's say 2/3rd of the guardians, then the previous proof is overwritten by the guardian to make sure the network is on the right path and has the right state.

This guardian proving should be really rare since there is this taiko token-based bond design i.e. validity bond which will be burnt if the proof is re-proven to be incorrect or they have the contestation bond i.e. if you contest a proof and it ends up that the original proof is correct and you are wrong, then your contestation bond is burnt.

This prevents people from spamming the network (unless they want to burn bonds, which is dumb).

Eventually, after the best tier i.e. highest tier proof (ZK proof) is solid and battle-tested and is really bug free, then the guardian provers should be gone since it's really just for the training wheel.

4. Benefits of Based Contestable Rollup

- Abstraction of special if-else code into a tier-based proof system makes the developers aware that the team cannot just shut down the chain uasing guardian prover and does not have control over it.
- Taiko has 3 types of bonds - validity bonds, contestation bonds and liveness bonds. We've spoken about the first two. Liveness bonds are basically, le's say, I have a prover off-chain and this prover is supposed to submit the proof within 15 minutes, then if the prover does not submit the proof in that time, then the prover's liveness bond is burnt.
- As an app dev, you can always change your config a long way. You can just use one layer-1 transaction to go from 100% optimistic to 100% ZK rollup.
- As ZK becomes more trustworthy, the team will slowly increase the % to ZK unitl they become fully ZK and remove the guardian prover.

5. Cooldown window for validity proofs

Since we know ZK is not fully trustworthy, let's give it 2-3 hours, so that if nobody challenges this ZK proof, it is final. So from that perspective, Taiko always allows validity proofs to be challenged and overwritten with a higher-tier proof. This adds security.

6. Another Benefit of Based Contestable Rollup

If someone says "I don't like the contestable feature", they can always configure their rollup with only one (top) tier. Then all proofs are not contestable and final. No validity bond nor contestable bond applies.

7. Taiko Mainnet Proofs

Initially: Optimistic => SGX => PSE zkEVM => Guardians

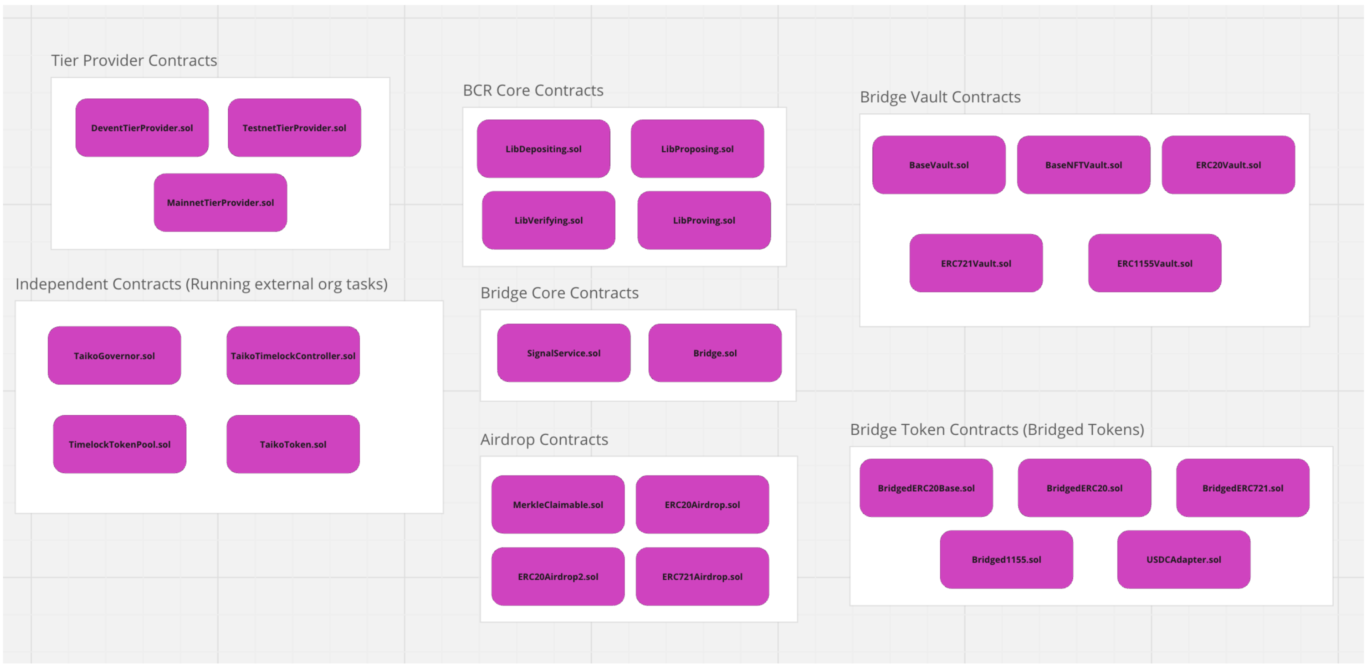
Later: Optimistic => SGX => PSE zkEVM => zkVM (risc0) => Guardians

End game: multiple zkVMs (Guardians removed)

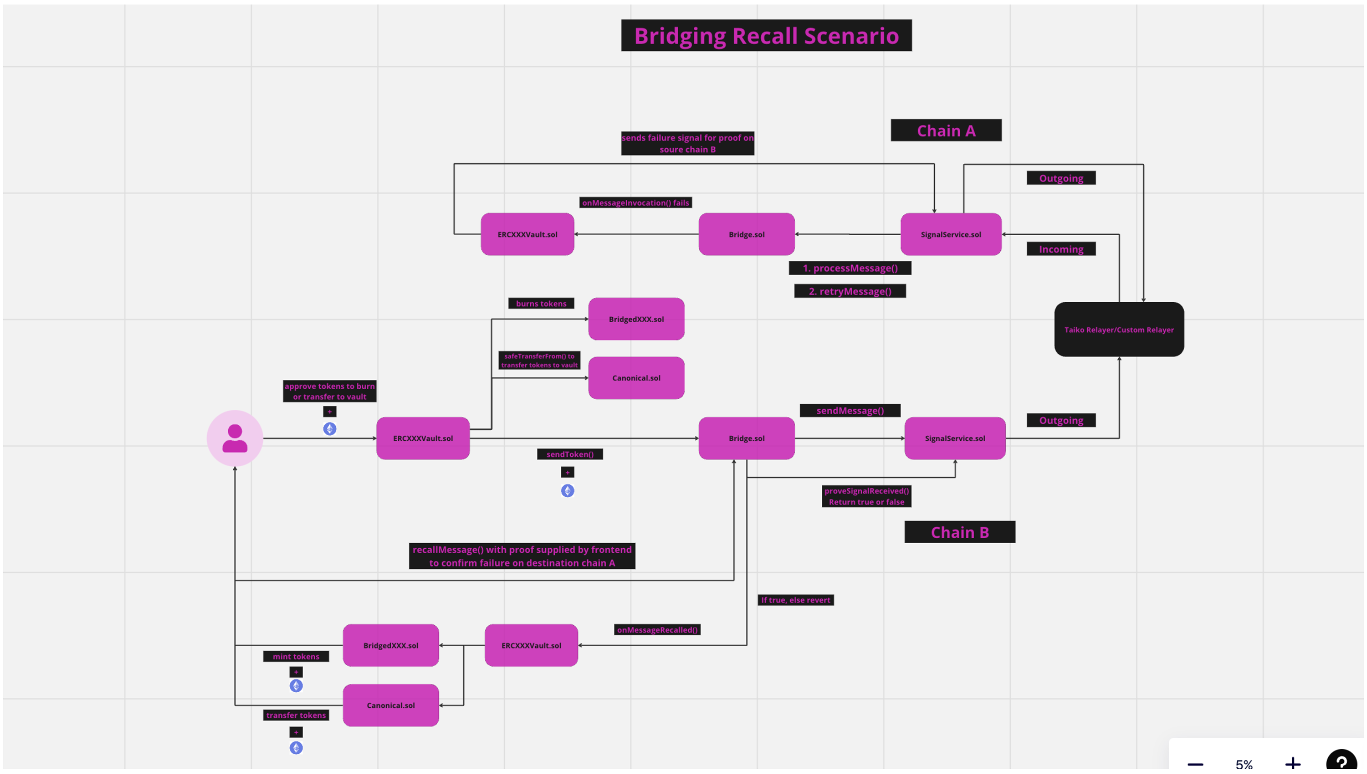
Chains supported

- Ethereum
- Taiko L2s/L3s

Grouping of Core Contracts



Recall specific scenario



Systemic Risks/Architecture-level weak spots and how they can be mitigated

There are a few risks associated with the protocol:

- The protocol does not have a robust on-chain fee estimation mechanism. On calling the on-chain functions, the relayers should provide the contracts with upto date prices for users or atleast maintain a default amount of gas to send across.
- The protocol would not work perfectly with swapping protocols. This is because the bridge includes invocation delays which can cause swaps to go outdated.
- There is an issue related to custom coinbase transfers which can create a risk among block proposers.