Kroma Security Audit

: Kroma, Optimistic Rollup with ZK Fault Proof

Sep 2, 2023

Revision 1.0

ChainLight@Theori

Theori, Inc. (“We”) is acting solely for the client and is not responsible to any other party. Deliverables are valid for and should be used solely in connection with the purpose for which they were prepared as set out in our engagement agreement. You should not refer to or use our name or advice for any other purpose. The information (where appropriate) has not been verified. No representation or warranty is given as to accuracy, completeness or correctness of information in the Deliverables, any document, or any other information made available. Deliverables are for the internal use of the client and may not be used or relied upon by any person or entity other than the client. Deliverables are confidential and are not to be provided, without our authorization (preferably written), to entities or representatives of entities (including employees) that are not the client, including affiliates or representatives of affiliates of the client.

Table of Contents

Executive Summary

Starting on July 10th, 2023, ChainLight of Theori audited the L1/L2 smart contracts and the validator (based on OP Stack) of the Kroma network for four weeks. Kroma is an Ethereum L2 and Optimistic Rollup with ZK Fault Proof.

We focused on identifying bugs that may allow theft of locked funds in the portal (L1 ↔ L2 bridge) contract and bugs that may allow finalization of an invalid L2 output root by always winning in interactive fault-proof process or avoiding detection of the invalid submission.

As a result, we found 26 (including seven informational issues) issues. The key findings are two critical issues: an issue in zkTrie validation that may lead to the theft of funds locked in the portal contract and an issue in the checkpointing feature of the validator that may lead to the invalid L2 output root being finalized and then ultimately lead to the theft of funds, and also four high severity issues that may allow unfair resolution of the fault-proof and/or theft of validators' funds.

Due to time constraints, we could not thoroughly review upstream implementations such as Geth, Optimism, and Scroll. zkEVM and ZK proof are out of the scope of this audit.

Audit Overview

Scope

Name	Kroma Security Audit
Target / Version	Git Repository (`kroma`): commit `b6dfbbc140d823318ffb826f076233596f24b7b4` (`v0.2.1`)
Application Type	Smart contracts Blockchain node (L2)
Lang. / Platforms	Smart contracts [Solidity] Blockchain node (L2) [Go]

Code Revision

N/A

Severity Categories

Severity	Description
Critical	The attack cost is low (not requiring much time or effort to succeed in the actual attack), and the vulnerability causes a high-impact issue. (e.g., Effect on service availability, Attacker taking financial gain)
High	An attacker can succeed in an attack which clearly causes problems in the service's operation. Even when the attack cost is high, the severity of the issue is considered "high" if the impact of the attack is remarkably high.
Medium	An attacker may perform an unintended action in the service, and the action may impact service operation. However, there are some restrictions for the actual attack to succeed.
Low	An attacker can perform an unintended action in the service, but the action does not cause significant impact or the success rate of the attack is remarkably low.
Informational	Any informational findings that do not directly impact the user or the protocol.

Status Categories

Status	Description
Confirm	ChainLight reported the issue to the vendor, and they confirm that they received.
Reported	ChainLight reported the issue to the vendor.
Fixed	The vendor resolved the issue.
Acknowledged	The vendor acknowledged the potential risk, but they will resolve it later.
WIP	The vendor is working on the patch.
Won't Fix	The vendor acknowledged the potential risk, but they decided to accept the risk.

Finding Breakdown by Severity

Category	Count	Findings
Critical	2	- `KROMA-005` - `KROMA-011`
High	4	- `KROMA-001` - `KROMA-002` - `KROMA-003` - `KROMA-007`
Medium	5	- `KROMA-004` - `KROMA-008` - `KROMA-017` - `KROMA-020` - `KROMA-024`
Low	7	- `KROMA-006` - `KROMA-009` - `KROMA-015` - `KROMA-016` - `KROMA-018` - `KROMA-019` - `KROMA-023`

Category	Count	Findings
Informational	**7**	• `KROMA-010` • `KROMA-012` • `KROMA-013` • `KROMA-014` • `KROMA-021` • `KROMA-022` • `KROMA-025`

Findings

Summary

#	ID	Title	Severity	Status
1	KROMA-001	`challengerTimeout()` does not check if the challenge is timed out	High	Fixed
2	KROMA-002	`proveFault()` can be called early to prove invalid states	High	Fixed
3	KROMA-003	`_callSecurityCouncil` uses a wrong L2 block number when `startingBlockNumber` is not zero	High	Fixed
4	KROMA-004	Upstream (Optimism) bugfixes should be applied	Medium	Fixed
5	KROMA-005	`ZKMerkleTrie` contract doesn't verify leaf node key	Critical	Fixed
6	KROMA-006	`ZKMerkleTrie` contract allows both empty and leaf nodes to appear	Low	Fixed
7	KROMA-007	`increaseBond()` doesn't check the caller	High	Fixed
8	KROMA-008	`ZKTrieHasher` allows `_valueHash` collision via `compressedFlags`	Medium	Fixed
9	KROMA-009	Asserter cannot collect pending bonds from challengers when the output root is finalized	Low	Fixed
10	KROMA-010	Nodes should be able to handle configuration changes of contracts	Informational	WIP

#	ID	Title	Severity	Status
11	`KROMA-011`	L1 reorg is not handled in the challenger	Critical	Fixed
12	`KROMA-012`	Incorrect usage of `Context.WithTimeout` may lead to resource leak	Informational	Fixed
13	`KROMA-013`	Typo in JSON Tag of Marshal Frame struct definition	Informational	Fixed
14	`KROMA-014`	Missing initialization for `ReentrancyGuardUpgradeable` in `MultiSigWallet` and `ValidatorPool` contract	Informational	Fixed
15	`KROMA-015`	Asserter can always reclaim bond by self-challenging	Low	Fixed
16	`KROMA-016`	`Burn.eth` uses an opcode not supported in the execution client	Low	Fixed
17	`KROMA-017`	Improvements for the next validator selection	Medium	Acknowledged
18	`KROMA-018`	Validator reward decay should have a lower bound	Low	Fixed
19	`KROMA-019`	Challenger can push an invalid `_segments[0]` when previous output Root is deleted	Low	Acknowledged
20	`KROMA-020`	Lack of validation for segments and proofs in `Colosseum.sol`	Medium	WIP
21	`KROMA-021`	Centralization risk of the security council for the fault-proof system	Informational	WIP
22	`KROMA-022`	`KromaPortal.GUARDIAN` should be a MultiSig wallet	Informational	Fixed

#	ID	Title	Severity	Status
23	`KROMA-023`	MultiSig wallet should confirm the transaction's content with its ID	Low	WIP
24	`KROMA-024`	Required bond amount for challengers should be limited	Medium	Fixed
25	`KROMA-025`	`createChallenge()` should have a check for L1 block hash to prevent challenger's loss due to L1 reorg	Informational	Fixed

#1 KROMA-001 challengerTimeout() does not check if the challenge is timed out

ID	Summary	Severity
KROMA-001	In Colosseum, the asserter can successfully execute challengerTimeout() any time it's their turn, regardless of whether the challenge is actually timed out.	High

Description

The external challengerTimeout function correctly checks that it's the asserter's turn, but doesn't check if the challenge status is CHALLENGER_TIMEOUT:

```
    function challengerTimeout(uint256 _outputIndex) external {
        Types.Challenge storage challenge = challenges[_outputIndex];
        _validateTurn(challenge);

        delete challenges[_outputIndex];
        emit Deleted(_outputIndex, block.timestamp);
    }
```

Note that _validateTurn() accepts any status where it is msg.sender's turn to act.

Impact

High

This issue allows the asserter to win any challenge, possibly allowing invalid L2 chain state roots to be committed.

Recommendation

The challenge status should be checked before deleting the challenge:

```
    function challengerTimeout(uint256 _outputIndex) external {
        Types.Challenge storage challenge = challenges[_outputIndex];
        _validateTurn(challenge);
        require(
          _challengeStatus(challenge) == ChallengeStatus.CHALLENGER_TIMEOUT,
          "Colosseum: invalid status"
        );

        delete challenges[_outputIndex];
        emit Deleted(_outputIndex, block.timestamp);
    }
```

Alternatively, since `_validateTurn()` internally computes `_challengeStatus(challenge)`, it could be modified to return the `ChallengeStatus` to allow further status checks without recomputing status, such as:

```
    function challengerTimeout(uint256 _outputIndex) external {
        Types.Challenge storage challenge = challenges[_outputIndex];
        ChallengeStatus status = _validateTurn(challenge);
        require(status == ChallengeStatus.CHALLENGER_TIMEOUT, "Colosseum: invalid status");

        delete challenges[_outputIndex];
        emit Deleted(_outputIndex, block.timestamp);
    }
```

Patch

Fixed

It is fixed as recommended.

#2 `KROMA-002` `proveFault()` can be called early to prove invalid states

ID	Summary	Severity
`KROMA-002`	In `Colosseum`, the asserter or challenger can successfully execute `proveFault()` any time it's their turn, regardless of whether the challenge is fully bisected or timed out.	High

Description

The external `proveFault` function correctly checks that it's the caller's turn to act, but doesn't check that the challenge status is either `READY_TO_PROVE` or `ASSERTER_TIMEOUT`:

```
function proveFault(
  ...
) external {
    Types.Challenge storage challenge = challenges[_outputIndex];

    _validateTurn(challenge);
    _validateOutputRootProof(
        _pos,
        challenge,
        _proof.srcOutputRootProof,
        _proof.dstOutputRootProof
    );
    ...
}
```

Note that `_validateTurn` accepts any status where it is `msg.sender`'s turn to act, and that `_validateOutputRootProof` doesn't fully validate the output roots if `_isAbleToBisect(challenge)`:

```
function _validateOutputRootProof(
  ..
```

```
    ) private view {
        bytes32 srcOutputRoot = Hashing.hashOutputRootProof(_srcOutputRootProof);
        bytes32 dstOutputRoot = Hashing.hashOutputRootProof(_dstOutputRootProof);

        // If asserter timeout, the bisection of segments may not have ended.
        // Therefore, segment validation only proceeds when bisection is not possible.
        if (!_isAbleToBisect(_challenge)) {
            require(
                _challenge.segments[_pos] == srcOutputRoot,
                "Colosseum: the source segment must be matched"
            );
            require(
                _challenge.segments[_pos + 1] != dstOutputRoot,
                "Colosseum: the destination segment must not be matched"
            );
        }

        require(
            _srcOutputRootProof.nextBlockHash == _dstOutputRootProof.blockHash,
            "Colosseum: the block hash must be matched"
        );
    }
```

Impact

High

Successfully executing `proveFault()` does not finalize the new state, but rather requests the security council to validate the proven state. As a result, challengers should be unable to finalize invalid states via this issue.

However, this issue also allows the asserter to call `proveFault()` during their turn, which will transition the challenge state to `PROVEN` with an `outputRoot` of their control, effectively allowing them to win any challenge.

Recommendation

The challenge status should be checked to be either `READY_TO_PROVE` or `ASSERTER_TIMEOUT`. Since `_validateTurn()` internally computes `_challengeStatus(challenge)`, it could be modified to return the `ChallengeStatus` to allow further status checks without recomputing status, such as:

```
    function proveFault(
      ...
    ) external {
        Types.Challenge storage challenge = challenges[_outputIndex];

        ChallengeStatus status = _validateTurn(challenge);
        require(
          status == ChallengeStatus.READY_TO_PROVE || status == ChallengeStatus.ASSERTER_TIMEOUT,
          "Colosseum: invalid status"
        );
        ...
```

Patch

Fixed

It is fixed as recommended.

#3 KROMA-003 _callSecurityCouncil uses a wrong L2 block number when startingBlockNumber is not zero

ID	Summary	Severity
KROMA-003	In Colosseum._callSecurityCouncil(), the L2 block number is calculated without considering the startingBlockNumber, causing a valid outputRoot from the challenger to be rejected by the security council.	High

Description

The _callSecurityCouncil() calculates L2 block number using uint128(_outputIndex * L2_ORACLE_SUBMISSION_INTERVAL):

```
    function _callSecurityCouncil(uint256 _outputIndex, bytes32 _outputRoo
t) private {
        // request outputRoot validation to security council
        SecurityCouncil(SECURITY_COUNCIL).requestValidation(
            _outputRoot,
            uint128(_outputIndex * L2_ORACLE_SUBMISSION_INTERVAL),
            abi.encodeWithSignature("approveChallenge(uint256)", _outputIn
dex)
        );
        emit Proven(_outputIndex, _outputRoot);
    }
```

When startingBlockNumber is non-zero, the calculated number is incorrect and would point to the different L2 block.

Since ValidateL2Output() uses the L2 block number to get the localOutputRoot, and compares the outputRoot submitted by the challenger to determine challenge approval, the function would return false in such cases.

```go
func (g *Guardian) ValidateL2Output(ctx context.Context, outputRoot eth.Bytes32, l2BlockNumber uint64) (bool, error) {
    g.log.Info("validating output...", "blockNumber", l2BlockNumber, "outputRoot", outputRoot)
    localOutputRoot, err := g.outputRootAtBlock(ctx, l2BlockNumber)
    if err != nil {
        return false, fmt.Errorf("failed to get output root at block %d: %w", l2BlockNumber, err)
    }
    isValid := bytes.Equal(outputRoot[:], localOutputRoot[:])
    return isValid, nil
}
```

Impact

High

Even if it's valid, the security council will not approve the challenge in such cases. So a wrong `outputRoot` may be finalized.

Recommendation

Modify the formula to include `startingBlockNumber` when calculating the L2 block number in `Colosseum._callSecurityCouncil()`.

Patch

Fixed

The calculation is removed, and the L2 block number from `L2OutputOracle` is used as a parameter for the security council.

#4 `KROMA-004` Upstream (Optimism) bugfixes should be applied

ID	Summary	Severity
`KROMA-004`	Many upstream (Optimism) bugfixes are not applied.	**Medium**

Description

The issues listed below should be addressed.

1. Incorrect gas calculation in `SafeCall.callWithMinGas()` https://github.com/ethereum-optimism/optimism/pull/5470

2. Error handling of re-entrant XDM messages can be improved https://github.com/ethereum-optimism/optimism/pull/5440 https://github.com/ethereum-optimism/optimism/pull/5444 https://github.com/ethereum-optimism/optimism/pull/5508

3. Add support for non-batched RPC calls https://github.com/ethereum-optimism/optimism/pull/5434

4. Finalize while syncing to avoid duplicate work https://github.com/ethereum-optimism/optimism/pull/5424

5. `unsafeL2Payloads` blocks when receiving a block prior to `unsafeL2Head` https://github.com/ethereum-optimism/optimism/issues/6092

6. Migtate L1BlockInfo MarshalBinary&UnmarshalBinary to writer&reader based API https://github.com/ethereum-optimism/optimism/pull/5400

7. Possibility of panic due to unchecked null pointer in the `pendingChannel` in the `TxFailed` function in `channel_manager.go` https://github.com/ethereum-optimism/optimism/pull/5417

8. Possibility of panic due to not using the latest version of `go-libp2p-pubsub` https://github.com/ethereum-optimism/optimism/pull/6032

9. Ensure proper draining of state during L2 reorganizations https://github.com/ethereum-optimism/optimism/pull/5536

10. Ensure `ResourceConfig.baseFeeMaxChangeDenominator` is larger than 1 https://github.com/ethereum-optimism/optimism/pull/5445

Impact

Medium

Recommendation

Apply patches from the Optimism repo.

Patch

Fixed

Code changes relevant to the listed issues have been merged from the upstream repo.

#5 KROMA-005 ZKMerkleTrie contract doesn't verify leaf node key

ID	Summary	Severity
KROMA-005	In `ZKMerkleTrie.get()`, a leaf node's `nodeKey` is not compared to the lookup key, enabling false inclusion proofs for non-existent keys.	Critical

Description

The Kroma L2 state is stored in a zkTrie (a type of binary trie using the Poseidon hash function for arithmetization efficiency). The `ZKMerkleTrie` solidity library is used to verify inclusion proofs in these tries, which is useful for accessing pieces of L2 state from a smart contract on L1.

The `ZKMerkleTrie.get()` method is used to validate a zkTrie proof (a list of nodes) and return the leaf node's value (and whether it exists in the trie). However, when validating the leaf node in a proof, the node's `nodeKey` is not compared to the key being looked up:

```
            } else if (currentNode.nodeType == NodeReader.NodeType.LEAF) {
                require(!exists, "ZKMerkleTrie: duplicated leaf node");
                exists = true;
                computedKey = _hashFixed3Elems(
                    bytes32(uint256(1)),
                    currentNode.nodeKey,
                    _valueHash(currentNode.compressedFlags, currentNode.va
luePreimage)
                );
                bytes32[] memory valuePreimage = currentNode.valuePreimage
;

                uint256 len = valuePreimage.length;
                assembly {
                    value := valuePreimage
                    mstore(value, mul(len, 32))
                }
```

```
                    ...
                }
```

Instead, the `exists` flag is always set to true. As a result, a valid non-existence proof (i.e. a proof of a non-matching leaf along the expected path) may be used to prove an invalid leaf in the zkTrie.

Impact

Critical

This bug likely affects every contract which uses the `ZKMerkleTrie`. `KromaPortal.sol` notably uses the zkTrie to validate withdrawals initiated from the L2. This issue enables an attacker to forge a withdrawal and steal user funds from the portal contract.

Recommendation

The `exists` flag should be set to `true` only if the `nodeKey` matches the expected `key`.

```
             } else if (currentNode.nodeType == NodeReader.NodeType.LEAF)
{
                 require(!exists, "ZKMerkleTrie: duplicated leaf node");
-                exists = true;
+                exists = currentNode.nodeKey == key;
                 computedKey = _hashFixed3Elems(
                     bytes32(uint256(1)),
                     currentNode.nodeKey,
                     _valueHash(currentNode.compressedFlags, currentNode.v
aluePreimage)
                 );
                 ...
             }
```

Patch

Fixed

It is fixed as recommended.

#6 `KROMA-006` `ZKMerkleTrie` contract allows both empty and leaf nodes to appear

ID	Summary	Severity
`KROMA-006`	In `ZKMerkleTrie.get()`, a proof which contains both an empty node and a leaf node may be accepted.	Low

Description

The `ZKMerkleTrie.get()` method is used to validate a zkTrie proof (a list of nodes) and return the leaf node's value (and whether it exists in the trie). The verification checks that at most one leaf node appears and that at most one empty node appears, but doesn't check that both don't appear.

Impact

Low

On its own, this issue has minimal impact, but it could have higher impact if combined with other issues in zkTrie verification.

Recommendation

When handling leaf nodes or empty nodes, both flags should be checked.

```
            } else if (currentNode.nodeType == NodeReader.NodeType.LEAF) {
-                require(!exists, "ZKMerkleTrie: duplicated leaf node");
+                require(!exists && !empty, "ZKMerkleTrie: duplicated terminal node");
                 ...
             }
```

```
             } else if (currentNode.nodeType == NodeReader.NodeType.EMPTY) {
-                require(!empty, "ZKMerkleTrie: duplicated empty node");
+                require(!empty && !exists, "ZKMerkleTrie: duplicated terminal node");
                 empty = true;
             }
```

Patch

Fixed

It is fixed as recommended.

#7 KROMA-007 increaseBond() doesn't check the caller

ID	Summary	Severity
KROMA-007	increaseBond() allows an unauthorized increase of the bond amount on behalf of any validator/challenger as if they challenged an L2 output root at the provided index.	High

Description

ValidatorPool.increaseBond() is intended to be called from Colosseum to handle the bond for the challenger. However, since it doesn't check the msg.sender, an attacker can increase the bond amount on behalf of any validator/challenger as if they challenged an L2 output root at the provided index.

```
    function increaseBond(address _challenger, uint256 _outputIndex) external {
        Types.Bond storage bond = bonds[_outputIndex];
        require(bond.expiresAt > 0, "ValidatorPool: the bond does not exist");

        uint128 bonded = bond.amount;
        _decreaseBalance(_challenger, bonded);
        bond.amount = bonded << 1;

        emit BondIncreased(_challenger, _outputIndex, bonded);
    }
```

A malicious validator can drain other validators' pool balance by forcing them to bond against the attacker's L2 output root that is valid and finalizing soon. Also, validators won't be able to challenge anything until they top up the pool balance and initiate a challenge before the attacker drains them again, potentially allowing an invalid L2 output root to be finalized. Since there is no upper bound in bond amount, it may also be possible to trick validators into depositing a significant amount of funds by submitting an invalid L2 output root with a large bond and then back-running the pool deposit with the exploit.

Impact

High

Theft of validator pool balances. An attacker can become the only priority validator by disqualifying other validators. Assets in the bridge can also be stolen by finalizing a forged L2 output root if honest validators/challengers and the contract admin fail to take countermeasures within the finalization period.

Recommendation

Check `msg.sender` in the `increaseBond()`. And consider limiting the size of the bond.

Patch

Fixed

It is fixed as recommended.

#8 `KROMA-008` `ZKTrieHasher` allows `_valueHash` collision via `compressedFlags`

ID	Summary	Severity
`KROMA-008`	A leaf node's `compressedFlags` field is not constrained, leading to leaf node hash collisions in `ZKTrieHasher._valueHash()`.	Medium

Description

In `ZKMerkleTrie.sol`, a zkTrie leaf node's hash is computed as follows:

```
                computedKey = _hashFixed3Elems(
                    bytes32(uint256(1)),
                    currentNode.nodeKey,
                    _valueHash(currentNode.compressedFlags, currentNode.va
luePreimage)
                );
```

where `_valueHash()` computes a hash of the leaf's value (`valuePreimage`) using the `compressedFlags` field as a hint for which value words to split and hash:

```
function _valueHash(uint32 _compressedFlags, bytes32[] memory _values)
        internal
        view
        returns (bytes32)
    {
        require(_values.length >= 1, "ZKTrieHasher: too few values for _va
lueHash");
        bytes32[] memory ret = new bytes32[](_values.length);
        for (uint256 i = 0; i < _values.length; ) {
            if ((_compressedFlags & (1 << i)) != 0) {
                ret[i] = _hashElem(_values[i]);
            } else {
                ret[i] = _values[i];
            }
            unchecked {
                ++i;
            }
        }
        if (_values.length < 2) {
            return ret[0];
        }
        return _hashElems(ret);
    }
```

In the zkTrie go module, the `compressedFlags` is set to a default value depending on the length of the leaf value:

```
    var vFlag uint32
    if val_sz == 160 {
        vFlag = 8
    } else if val_sz == 128 {
        vFlag = 4
    } else {
        vFlag = 1
    }
```

However, when verifying proofs on chain, the `compressedFlags` are fully controlled by the user and are not constrained, leading to hash collisions.

An easy example collision, notice that for any `x`, `_valueHash(0, [x]) == x`. Thus if some leaf has actual flags `f` and value preimage `v`, an attacker can set `compressedFlags = 0` and `valuePreimage = _valueHash(f, v)` to cause a collision.

Impact

Medium

This bug doesn't seem to lead to exploitable scenarios in the current Kroma contracts. However, it could impact any other zkTrie state proofs which get introduced.

Recommendation

The `compressedFlags` value for each value length should be more clearly defined, and `ZKTrieHasher._valueHash()` should require that these flags match the expected value.

Patch

Fixed

It is fixed as recommended.

#9 KROMA-009 Asserter cannot collect pending bonds from challengers when the output root is finalized

ID	Summary	Severity
KROMA-009	If the L2 output root that was the target of the challenge is finalized, the asserter cannot collect a bond for the challenge that failed due to timeout.	Low

Description

```
function challengerTimeout(uint256 _outputIndex, address _challenger)
    external
    outputNotFinalized(_outputIndex)
{
// ...
}

modifier outputNotFinalized(uint256 _outputIndex) {
    require(
        !L2_ORACLE.isFinalized(_outputIndex),
        "Colosseum: cannot progress challenge process about already finali
zed output"
    );
    _;
}
```

Because the `challengerTimeout()` has the `outputNotFinalized` modifier, the asserter cannot receive a bond even if the challenger has timed out in the case of a challenge for the finalized L2 output root.

Impact

Low

This is not a serious problem because the asserter has sufficient time, but not receiving a bond that should be received is undesirable.

Recommendation

In the `CHALLENGER_TIMEOUT` state, it should be allowed to seize the bond regardless of whether the corresponding L2 output root has been finalized or not.

Patch

Fixed

Fixed as recommended.

#10 `KROMA-010` Nodes should be able to handle configuration changes of contracts

ID	Summary	Severity
`KROMA-010`	Some configuration values used by the node are fetched from the contract. However, if these values are defined as constants, their values are only fetched during initialization, despite the possibility of change.	Informational

Description

The configuration values are fetched from the contract during the node's initialization, but there is no feature for later updates. These values are defined as `immutable`. However, they may undergo changes because the L1 contracts of Korma are upgradable. It is also mentioned in the comment that these values may be changed. Thus, there is an issue where configuration changes made through contract upgrades are not reflected until the node is restarted.

`kroma-network/kroma/components/validator/challenger.go`

```
submissionInterval, err := l2ooContract.SUBMISSIONINTERVAL(callOpts)
// ...
finalizationPeriodSeconds, err := l2ooContract.FINALIZATIONPERIODSECONDS(c
allOpts)
// ...
l2BlockTime, err := l2ooContract.L2BLOCKTIME(callOpts)
```

`kroma-network/kroma/components/validator/l2_output_submitter.go`

```
l2BlockTime, err := l2ooContract.L2BLOCKTIME(callOpts)
// ...
submissionInterval, err := l2ooContract.SUBMISSIONINTERVAL(callOpts)
```

`kroma-network/kroma/packages/contracts/contracts/L1/L2OutputOracle.sol`

```
/**
 * @notice The interval in L2 blocks at which checkpoints must be submitte
d. Although this is
 *         immutable, it can safely be modified by upgrading the implement
ation contract.
 */
uint256 public immutable SUBMISSION_INTERVAL;
```

Impact

Informational

If the nodes are not restarted after a contract upgrade, they may malfunction. However, the Korma team has stated that the configuration values will not be changed in the near future.

Recommendation

The configuration values should be polled, or the node should be modified to operate in accordance with the new values by checking the contract upgrade event.

Patch

WIP

The "safely" part has been removed from the explanation in the comment stating that they can be modified, and an explanation that the node must be restarted when modified has been added. The node code will also be modified.

#11 KROMA-011 L1 reorg is not handled in the challenger

ID	Summary	Severity
KROMA-011	The challenger receives data from the latest block in L1, validates it, and saves the checkpoint to avoid redundant validation. However, it can result in an unchecked output root if reorg occurs in L1 because the challenger ignores the blocks older than the checkpoint.	Critical

Description

In the challenger, if an L2 output root submit event occurs, the outputs from the last checked output index (checkpoint) up to the output specified in the event are checked. Therefore, if the submitted transaction for an output index at or before the checkpoint is invalidated due to L1 reorg, the resent output will not be checked.

While an attacker performing validation normally as an validator, the attacker could detect that the L2 output root submit transaction they sent was invalidated by an accidental reorg. Subsequently, the attacker can send a manipulated output root and, after waiting for its finalization, drain the bridge's entire assets. (Reorg can occur under normal circumstances for various reasons, and it is also possible for the attacker to directly trigger reorg if they are the L1 validators.)

`kroma-network/kroma/components/validator/challenger.go`

```go
c.checkpoint = new(big.Int).Sub(nextOutputIndex, common.Big1)

if err := c.scanPrevOutputs(c.ctx); err != nil {
```

```go
func (c *Challenger) subscribeL2OutputSubmitted(ctx context.Context) {
    defer c.wg.Done()

    for {
        select {
        case ev := <-c.l2OutputSubmittedEventChan:
            c.log.Info("watched output submitted event", "l2BlockNumber",
ev.L2BlockNumber, "outputRoot", ev.OutputRoot, "outputIndex", ev.L2OutputIndex)
            // validate all outputs in between the checkpoint and the current outputIndex
            for i := new(big.Int).Add(c.checkpoint, common.Big1); i.Cmp(ev.L2OutputIndex) != 1; i.Add(i, common.Big1) {
                c.wg.Add(1)
                go c.handleOutput(ctx, new(big.Int).Set(i))
            }
            c.checkpoint = ev.L2OutputIndex
            c.metr.RecordChallengeCheckpoint(c.checkpoint)
        case <-ctx.Done():
            return
        }
    }
}
```

Impact

Critical

If the challenger node is restarted in the middle of the attack, a scan for the previous event will be performed again, causing the attack to fail; otherwise, the likelihood of detection is very low. Although the attack opportunity is limited, this is a satisfactory condition if the attacker waits long enough, and they can eventually finalize the invalid L2 output root and drain all the bridge's assets.

Recommendation

It is recommended to either handle the invalidation of data that have already been checked due to L1 reorg or receive data only from blocks that were finalized.

Patch

Fixed

If an event is received for an L2 output index equal to or less than the checkpoint, it is presumed that reorg has occurred, and the output specified in the event is checked regardless of the checkpoint.

#12 `KROMA-012` Incorrect usage of `Context.WithTimeout` may lead to resource leak

ID	Summary	Severity
`KROMA-012`	In some cases, context management using defer is not implemented correctly when contract call or RPC call is made via Goroutine by the validator or node of components.	Informational

Description

This problem mainly arises when execution time is constrained by `NetworkTimeout` using the function `context.WithTimeout`. If an exception occurs in a function that uses `context` but does not correctly implement resource cleanup via `defer cancel()`, a resource leak occurs because `cancel()` is not called.

Example)

```
func (c *Challenger) BuildSegments(ctx context.Context, turn uint8, segStart, segSize uint64) (*chal.Segments, error) {
    cCtx, cCancel := context.WithTimeout(ctx, c.cfg.NetworkTimeout)
    sections, err := c.colosseumContract.GetSegmentsLength(utils.NewSimpleCallOpts(cCtx), turn)
    cCancel() // @audit: This may not be executed
    if err != nil {
        return nil, fmt.Errorf("unable to get segments length of turn %d: %w", turn, err)
    }
// ...
```

Example of a function that has this problem: `components/batcher/batch_submitter.go`

```
func (b *BatchSubmitter) loadBlockIntoState(ctx context.Context, blockNumber uint64) (eth.BlockID, error) {
```

`components/node/p2p/sync.go`

```
func (s *SyncClient) mainLoop() {
```

`components/node/p2p/sync.go`

```
func (s *SyncClient) doRequest(ctx context.Context, id peer.ID, n uint64)
error {
```

`components/node/p2p/sync.go`

```
func (srv *ReqRespServer) HandleSyncRequest(ctx context.Context, log log.L
ogger, stream network.Stream) {
```

`components/node/rollup/driver/state.go`

```
func (d *Driver) eventLoop() {
```

`components/validator/guardian.go`

```
func (g *Guardian) ConfirmTransaction(ctx context.Context, transactionId *
big.Int) (*types.Transaction, error) {
```

`components/validator/guardian.go`

```
func (g *Guardian) processOutputValidation(ctx context.Context, event *bin
dings.SecurityCouncilValidationRequested) {
```

`components/validator/l2_output_submitter.go`

```
func NewL2OutputSubmitter(ctx context.Context, cfg Config, l log.Logger, m
 metrics.Metricer) (*L2OutputSubmitter, error) {
```

`components/validator/l2_output_submitter.go`

```go
func (l *L2OutputSubmitter) fetchBlockNumbers(ctx context.Context) (*big.Int, *big.Int, error) {
```

Impact

Informational

If exceptions occur frequently, the `context` can continuously leak, potentially leading to a gradual increase in node memory usage and then availability issues.

Recommendation

Parts with inadequate context management among functions within components (batcher, node, validator) should be modified as shown below. (No modifications are necessary when calling a function where there is absolutely no possibility of an exception or when the process is terminated because the exception is not handled.)

```
func (c *Challenger) BuildSegments(ctx context.Context, turn uint8, segStart, segSize uint64) (*chal.Segments, error) {
    cCtx, cCancel := context.WithTimeout(ctx, c.cfg.NetworkTimeout)
+   defer cCancel()
    sections, err := c.colosseumContract.GetSegmentsLength(utils.NewSimpleCallOpts(cCtx), turn)
-   cCancel()
    if err != nil {
        return nil, fmt.Errorf("unable to get segments length of turn %d: %w", turn, err)
    }
```

If the code to be modified is inside a loop, a function should be created to prevent the continuous accumulation of `defer` in the stack.

```
func slowOperationWithTimeout(ctx context.Context) (Result, error) {
    ctx, cancel := context.WithTimeout(ctx, 100*time.Millisecond)
    defer cancel()  // releases resources if slowOperation completes before timeout elapses
    return slowOperation(ctx)
}
```

Patch

Fixed

All functions are fixed as recommended, including functions not mentioned in the Description.

#13 `KROMA-013` Typo in JSON Tag of Marshal Frame struct definition

ID	Summary	Severity
`KROMA-013`	A typo in the JSON Tag of a struct definition should be fixed.	Informational

Description

The correct JSON tag is in the form of `json:"full_name"`, but there is `'json:"is_last"` in the Frame struct's definition. While it does not cause any problems during runtime, it may become problematic (e.,g, linter error, test error, etc.) and should be fixed.

Impact

Informational

Recommendation

```
type Frame struct {
  ID          ChannelID `json:"id"`
  FrameNumber uint16    `json:"frame_number"`
  Data        []byte    `json:"data"`
- IsLast      bool      `'json:"is_last"`
+ IsLast      bool      `json:"is_last"`
  }
```

The JSON Tag value of the `IsLast` field needs to be updated to `json:"is_last"`.

Patch

Fixed

It is fixed as recommended.

#14 `KROMA-014` Missing initialization for `ReentrancyGuardUpgradeable` in `MultiSigWallet` and `ValidatorPool` contract

ID	Summary	Severity
`KROMA-014`	A call to the initializer of the `ReentrancyGuard` is missing for some contracts.	Informational

Description

`__ReentrancyGuard_init_unchained()` is not called in `MultiSigWallet.initialize()` and `ValidatorPool.initialize()`, so the `ReentrancyGuardUpgradeable` is not initialized.

Impact

Informational

The `ReentrancyGuardUpgradeable` contract of the current version works even if not initialized. However, it can become problematic with a change in the initialization function or other logic due to a library upgrade.

Recommendation

`__ReentrancyGuard_init_unchained()` should be called from `MultiSigWallet.initialize()` and `ValidatorPool.initialize()`.

Patch

Fixed

It is fixed as recommended.

#15 `KROMA-015` Asserter can always reclaim bond by self-challenging

ID	Summary	Severity
`KROMA-015`	The bond may not be lost even if a wrong L2 output root has been submitted because the attacker can perform both the challenger and asserter roles.	Low

Description

If the attacker submits a wrong L2 output root as an asserter and a challenge is initiated upon discovery, the attacker can reclaim the bond by taking on the challenger role with another account. This is possible because the asserter can decide which challenge to quickly respond to, and only the challenger who succeeds in a challenge in the first obtains the asserter's bond.

Impact

Low

The attacker can check the number of active challengers during preparation for another attack without risking the loss of bonds. If there are no active challengers, the incorrect L2 output root can be confirmed, and the assets of the bridge can be stolen.

Recommendation

To prevent collusion, a portion of the bonds won by the challenger or asserter should be burned.

Patch

Fixed

If the challenge is successful, 20% of the bonds will be deducted. Currently, the bonds are not burned but held by the security council.

#16 KROMA-016 Burn.eth uses an opcode not supported in the execution client

ID	Summary	Severity
KROMA-016	The Burn library uses the selfdestruct opcode, and the library is non-functional since this opcode has been removed from the execution client.	Low

Description

The Burn library creates a Burner contract, and the Burner uses the selfdestruct opcode.

kroma-network/kroma/packages/contracts/contracts/libraries/Burn.sol

```
library Burn {
    /**
     * Burns a given amount of ETH.
     *
     * @param _amount Amount of ETH to burn.
     */
    function eth(uint256 _amount) internal {
        new Burner{ value: _amount }();
    }
// ...
}

// ...

contract Burner {
    constructor() payable {
        selfdestruct(payable(address(this)));
    }
}
```

However, the `selfdestruct` opcode has been removed from the execution client because zkEVM does not support it.

Example)

```
    // [Scroll: START]
    // NOTE: SELFDESTRUCT is disabled in Kroma. This is not meant to disable
    // forever this opcode. Once zkevm spec can cover it, we need to re-enable it.
    // jt[SELFDESTRUCT].constantGas = params.SelfdestructGasEIP150
    // jt[SELFDESTRUCT].dynamicGas = gasSelfdestructEIP2929
    // [Scroll: END]
```

Impact

Low

The function `Burn.eth` cannot be used.

Recommendation

Burn should be implemented using a method other than `selfdestruct` (e.g., transfer to a burn address)

Patch

Fixed

It is fixed as recommended. (transfer to the zero address)

#17 `KROMA-017` Improvements for the next validator selection

ID	Summary	Severity
`KROMA-017`	Suggestion for improvement of the priority validator selection process.	Medium

Description

```
/**
 * @notice Updates next priority validator address.
 *
 * @param _outputRoot The L2 output of the checkpoint block.
 */
function _updatePriorityValidator(bytes32 _outputRoot) private {
    uint256 len = validators.length;
    if (len > 0) {
        // TODO(pangssu): improve next validator selection
        uint256 validatorIndex = uint256(
            keccak256(abi.encodePacked(_outputRoot, block.number, block.co
inbase))
        ) % len;

        nextPriorityValidator = validators[validatorIndex];
    } else {
        nextPriorityValidator = address(0);
    }
}
```

Currently, `_outputRoot`, `block.number`, and `block.coinbase` are used for the selection of the priority validator. If it is possible to predict the rest of the values at the time of selection, a validator can affect the L2 output root so that the desired validator is selected later.

The `block.number` is relatively easy to predict because it is correlated with the time when the priority validator is selected. The `block.coinbase` is more difficult to predict because the ETH validator to be selected must be known. However, the validator list is public, and we can use that information because the higher the amount of staked ETH, the more likely it is to be selected.

Furthermore, even if the random number generation is fair, registering many validators can be a problem because it increases the probability of being selected.

Impact

Medium

The attacker can gain more validator rewards by increasing the probability of becoming a priority validator, weakening the willingness of other validators to operate the node.

Recommendation

We need to make it more challenging to predict by adding `block.prevrandao` and `blockhash(block.number - 1)` to the items to be hashed. Furthermore, because it is difficult to block registering many validators with one node, it is desirable to allow all validators to have a proportionally higher chance of being selected if they lock a large amount without modifying the node. Thus, we should allow validators to lock ETH and select the next validator at a weighted random proportionate to the amount of ETH in the locked state. (It is recommended to increase the lock duration to twice the validator selection cycle.)

Although this is not an ideal method, we can also allow only the node operators who have been approved by the Kroma team to sign up as validators.

Patch

Acknowledged

Prediction has become more challenging due to the inclusion of `block.difficulty` and `blockhash(block.number - 1)` in the hash items. However, staking was not implemented due to the launch schedule.

#18 KROMA-018 Validator reward decay should have a lower bound

ID	Summary	Severity
KROMA-018	The validator reward decreases as rounds progress and may eventually reach zero, but there should be a minimum value to incentivize submissions, even if they are delayed.	Low

Description

The priority validator can submit the L2 output root for 30 minutes; after this, any validator can submit it. The validator reward gradually decreases after 40 minutes and reaches zero if the output root is not submitted within an hour. In general circumstances, there is an incentive to submit the output root quickly because a 100% reward can be obtained for quick submissions.

However, if the output root is not submitted before the reward reaches zero due to an issue, the validator is less motivated to submit it because it will not receive any reward while consuming gas. (For example, if the proposer is halted for an extended period, there may be many L2 output roots with the minimum reward due to the disparity between the L2 timestamp calculated from the L2 block number and the L1 timestamp.)

Impact

Low

Users' withdrawal waiting times may increase when the L2 output root is not submitted, causing inconvenience. However, the current node code mandates the submission of the L2 output root even when the reward is zero. Therefore, it will be submitted unless the node code is modified, but node operators will be unhappy as it consumes gas without any reward.

Recommendation

There should be a limit on the reduction of the validator reward so that even if it decreases to the minimum, the gas used will still be compensated.

Patch

Fixed

The reduction of the validator reward was removed because it is considered unfair for the validator's reward to be reduced due to the proposer's delay.

#19 KROMA-019 Challenger can push an invalid _segments[0] when previous outputRoot is deleted

ID	Summary	Severity
KROMA-019	In Colosseum.createChallenge(), the challenger can set an arbitrary _segments variable if the previous outputRoot has been deleted. The challenger can win the interactive fault proof by generating a forceful ASSERTER_TIMEOUT if the value of _segments[0] is set to an invalid value.	Low

Description

If the previous output has been deleted in Colosseum.createChallenge(), the validity of _segments[0] is not checked. Therefore, a malicious challenger can set arbitrary segments.

```
    function createChallenge(uint256 _outputIndex, bytes32[] calldata _segments)
        external
        outputNotFinalized(_outputIndex)
    {
        ...
        // If the previous output has been deleted, the first segment will not be compared with the previous output.
        if (prevOutput.outputRoot == DELETED_OUTPUT_ROOT) {
            _validateSegments(TURN_INIT, _segments[0], targetOutput.outputRoot, _segments);
        } else {
            _validateSegments(TURN_INIT, prevOutput.outputRoot, targetOutput.outputRoot, _segments);
        }
        ...
    }
```

If a malicious challenger creates a challenge by setting `_semgents[0]` to an invalid value, the asserter will not proceed with `bisect()` normally. In `Colosseum.bisect()`, starting an interactive fault proof challenge on `_segments[0]` are not allowed, and if `_segments[0]` is invalid, `selectFaultPosition()` in Challenger.go, which is executed by the asserter, will return a value of -1.

```go
func (c *Challenger) selectFaultPosition(ctx context.Context, segments *ch
al.Segments) (*big.Int, error) {
    for i, blockNumber := range segments.BlockNumbers() {
        output, err := c.OutputAtBlockSafe(ctx, blockNumber)
        if err != nil {
            return nil, err
        }

        if !bytes.Equal(segments.Hashes[i][:], output.OutputRoot[:]) {
            return big.NewInt(int64(i) - 1), nil
        }
    }

    return nil, errors.New("failed to select fault position")
}
```

Because the position is returned with a -1 value, `_pos` of `bisect(uint256 _outputIndex, address _challenger, uint256 _pos, bytes32[] calldata _segments)` does not have a type match with uint256 type, and consequently `bisect()` in Challenger.go returns an error. The asserter does not perform any action, and `ASSERTER_TIMEOUT` occurs after the elapse of a time equal to `BISECTION_TIMEOUT`.

If a malicious challenger has set all segments except `_segments[0]` to valid values when running `createChallenge()`, they can win the challenge by generating a `_pos` value, which is not 0, and `outputRootProof` (stateRoot, messagePasserStorageRoot, etc.) and `publicInput` (blockHash, parentHash, timestamp, etc) for the L2 block corresponding to `segments[_pos]` and the next L2 block, and performing `proveFault()`. In this case, the corresponding output is deleted, and the malicious challenger takes the asserter's bond.

In the above scenario, the asserter could interact directly with the contract to avoid `ASSERTER_TIMEOUT`, ignoring the invalid `segments[0]` and continuing to call `bisect()`, eventually forcing the challenger to perform proofs for the block that matches `segments[0]` and the next block through `proveFault()`. However, `proveFault()` currently does not perform additional data validation to ensure that the data required for the proof matches the L2 block number of the actual disputed position. Therefore, the challenger can perform the proof using `PublicInputProof` and `segments[0]` that do not match the actual L2 state.

Impact

Low

A malicious challenger can take the asserter's bond and trigger a delay attack by continuously deleting the `outputRoot`. However, as the security council monitors for events in which valid outputRoot is deleted and calls `dismissChallenge()`, the attack is invalidated and the attacker loses the bond used in the attack.

In the future, if the role of the security council is reduced and the `dismissChallenge()` function is removed, this attack could pose a bigger problem.

Recommendation

The recommendations are the same as those of [KROMA-020].

Patch

Acknowledged

In the future, the fault proof system will be improved such that proofs can be performed using the L2 transaction batch sent to L1; however, for now, the security council will respond to the attack.

#20 `KROMA-020` Lack of validation for segments and proofs in `Colosseum.sol`

ID	Summary	Severity
`KROMA-020`	`Colosseum.proveFault()` can prove whether the state transition is valid; however, it relies on the previous `outputRoot` to verify whether the data needed to compute the state transition provided by the challenger are L2's data. If the previous `outputRoot` is deleted or wrong, it is possible to attack to delete a valid `outputRoot` or finalize an invalid `outputRoot`.	Medium

Description

Unlike Arbitrum's state transformation function, which considers TXs batched to the L1 delayed Inbox as input, and Optimism's Cannon, which uses the L1 blockhash to validate the state transformation function's input, the current `Colosseum.proveFault()` relies on `segments[_pos]` to validate that the `PublicInputProof` was derived from L2 TXs batched to Ethereum.

When `segments[_pos]` is valid in `proveFault()`

When `segments[_pos]` is valid, `_proof.srcOutputRootProof` is checked by `_validateOutputRootProof()` and thus cannot be manipulated. Because `_proof.srcOutputRootProof.nextBlockHash == _Hashing.hashBlockHeader(_publicInput, _rlps)`, `publicInput` and `rlps` also cannot be manipulated. The values of `dstOutputRootProof.stateRoot` and `dstOutputRootProof.blockHash` also cannot be manipulated due to the following conditional statements: `validateOutputRootProof()`'s `require(_srcOutputRootProof.nextBlockHash == _dstOutputRootProof.blockHash)` and `_validatePublicInput()`'s `require(_publicInput.stateRoot == _dstOutputRootProof.stateRoot);`.

Because it is guaranteed that `dstOutputRootProof.stateRoot` is valid, it is guaranteed by `_validateWithdrawalStorageRoot()` that `dstOutputRootProof.messagePasserStorageRoot` cannot be manipulated.

Finally, because the `dstOutputRootProof.version` is allowed only when `hashOutputRootProof` is `bytes32(0)`, it cannot be manipulated.

However, the assumption that `dstOutputRootProof` cannot be manipulated is guaranteed in chains from the condition that `requires(_proof.srcOutputRootProof.nextBlockHash == _Hashing.hashBlockHeader(_publicInput, _rlps));` if `srcOutputRootProof` is manipulated, it is also possible to manipulate `dstOutputRootProof`.

When `segments[_pos]` is invalid in `proveFault()`

When the previous `outputRoot` has been deleted, as in [KROMA-019], the challenger can create a challenge by setting arbitrary `segments[0]`. This causes `ASSERTER_TIMEOUT`, which allows the following attack when a malicious challenger performs `proveFault()`.

If `_pos` is specified as 0 when calling `proveFault()`, the challenger can submit a `srcOutputRoot` that is matched to `segments[0]`. However, because the challenger can create arbitrary `segments[0]` when performing `createChallenge`, they can also manipulate `srcOutputRootProof`. Consequently, the challenger can call the function with whatever values they wish for not only `srcOutputRootProof` but also all other input values, such as `dstOutputRootProof` and `publicInput`, and always win the challenge.

The bigger problem is that the public input used by the challenger in `proveFault()` is registered in the `verifiedPublicInputs` mapping and cannot be used again. If the public input in this case is required to prove another invalid output index, the challenge for another output index cannot be performed normally. This can lead to the finalization of an invalid L2 output root. For example, the following scenario is possible.

1. Suppose an attacker submits an invalid `stateRoot` at `outputIndex` 3 (Interval for L2 block 3600 ~ 5400 when `SUBMISSION_INTERVAL` is 1800 blocks) and manipulates the 4000th L2 block. Here, the challenger must perform an interactive fault proof for the 4000th `stateRoot` through `createChallenge()` and `proveFault()`. Multiple challengers start to dispute.
2. Suppose `outputIndex` 1 also has an invalid `outputRoot`, and it is deleted through a challenge. Because the previous `outputRoot` has been deleted, the attacker can perform an interactive fault proof with arbitrary values of segments in the challenge for `outputIndex` 2. Using this, the attacker calls `createChallenge()` by putting the `outputRoot` for L2 block 3999 in `segments[0]` and `outputRoot` for L2 block 4000 in `segments[1]`.
3. If the asserter for `outputIndex` 2 is also another address of the attacker, `bisect()` is performed quickly to move the challenge to the `PROVE_READY` state. Even if the asserter is not the attacker, `Challenger.go`'s `selectFaultPosition()` returns a value of -1 when `segments[0]` is invalid, preventing `bisect()` from proceeding properly. Consequently, `ASSERTER_TIMEOUT` occurs.

4. The attacker performs `proveFault()` for `outputIndex` 2. Here, it is called with a value of 1 for `_pos`, and the interactive fault proof is performed for `segments[0]` (`outputRoot` for L2 block 3999) and `segments[1]` (`outputRoot` for L2 block 4000). By submitting a public input for L2 block 4000, the attacker ensures that the `publicInput` used here cannot be used in the future through `verifiedPublicInputs[publicInputHash] = true`.
5. When other challengers create challenges for `outputIndex` 3 and perform `proveFault()` for L2 block number 4000, they must use a `publicInput` value that has been already used. This leads to a revert. Other challengers lose the disputes because of this, and after seven days, the invalid `outputRoot` is finalized. In this case, a malicious validator can drain all assets in the bridge.

If a valid `outputRoot` is deleted, the security council node detects the event and calls the `dismissChallenge()` to restore the deleted `outputRoot`, and the malicious attacker may lose the bond. However, even in this case, `verifiedPublicInputs[publicInputHash] = true`, and if the `publicInput` is a value required in a challenge for another invalid output, `proveFault()` for that `outputRoot` is still impossible. In this case, because no additional event occurs, there is a risk that an invalid `outputRoot` will be finalized unless the security council calls `forceDeleteOutput()` to delete that `outputRoot`.

Impact

Medium

A malicious challenger can win an interactive fault proof over a valid `outputRoot` based on an invalid proof and take the asserter's bond. In this case, the security council can invalidate the attack by calling `dismissChallenge()`.

For an attack that uses a `publicInput` in advance, an invalid `outputRoot` can be finalized due to failure to perform a challenge on the invalid `outputRoot`, and the attacker can use the invalid `outputRoot` to drain the assets in the bridge. In this case, the security council can invalidate the attack by calling `forceDeleteOutput()`.

Recommendation

In the current fault proof system, a valid fault proof for the next `outputRoot` is possible only when the previous `outputRoot` is determined to be a valid value. However, for now, in the case of succeeding in challenge, this assumption is broken because the previous `outputRoot` is deleted with `bytes(0)`.

In the short term, when succeeding in the challenge, the `outputRoot` should not be overwritten with `bytes32(0)`. Instead, that `outputRoot` and all subsequent `outputRoots` should be deleted as in Optimism, or a valid `outputRoot` should be inserted via security council or other means.

In the long term, the fault proof system should be changed to perform proofs via L2 transactions batched to L1, upgrading from the current method that relies on `srcOutputRootProof.nextBlockHash`.

Patch

WIP

Revision has been made to delete what is written in `verifiedPublicInputs` when the security council performs `dismissChallenge` such that the valid public input can be reused in the proper challenge.

It is planned to improve the fault proof system such that the proof can be performed using the L2 transaction batch sent to L1; however, for now, the security council will respond to the attack.

#21 `KROMA-021` Centralization risk of the security council for the fault-proof system

ID	Summary	Severity
`KROMA-021`	There are several potential risks in the fault-proof system of Kroma (`Colosseum.sol`), but the security council can handle these risks. However, there is a centralization risk for the security council, and bridged assets can be stolen if the security council is compromised.	Informational

Description

In `Colosseum.sol`, there are several potential risks that the Security Council can defend against:

1. ZK-proving is not possible due to an undeniable bug
2. [KROMA-020] Lack of validation for segments and proofs in `Colosseum.sol`
3. [KROMA-019] Challenger can push an invalid `_segments[0]` when previous outputRoot is deleted

The security council can defend the above attacks by calling `dismissChallenge()` and `forceDeleteOutput()`.

In `dismissChallenge()`, the security council can put an arbitrary `outputRoot` value in `DELETED_OUTPUT_ROOT` before finalization. If an attacker compromises the security council, the attacker can put an invalid outputRoot during the creation period (`CREATION_PERIOD_SECONDS`, one day before finalization) and steal all bridged assets after the `outputRoot` is finalized. In this case, validators cannot even create a new challenge due to the limitation of the creation period.

```
    function dismissChallenge(
        uint256 _outputIndex,
        address _challenger,
        address _asserter,
        bytes32 _outputRoot
    ) external onlySecurityCouncil {
        require(
            _outputRoot != DELETED_OUTPUT_ROOT,
            "Colosseum: cannot rollback output to zero hash"
        );
        require(
            L2_ORACLE.getL2Output(_outputIndex).outputRoot == DELETED_OUTP
UT_ROOT,
            "Colosseum: only can rollback if the output has been deleted"
        );

        // Rollback output root.
        L2_ORACLE.replaceL2Output(_outputIndex, _outputRoot, _asserter);

        emit ChallengeDismissed(_outputIndex, _challenger, block.timestamp
);
    }
```

In `forceDeleteOutput()`, the security council can delete a specific `outputRoot`. Suppose an attacker compromises the security council. In that case, the attacker can continuously delete the `outputRoot` to prevent L2 users from withdrawing and take the collateral of the asserter who submitted the `outputRoot`.

```
    function forceDeleteOutput(uint256 _outputIndex)
        external
        onlySecurityCouncil
        outputNotFinalized(_outputIndex)
    {
        // Delete output root.
        L2_ORACLE.replaceL2Output(_outputIndex, DELETED_OUTPUT_ROOT, SECUR
ITY_COUNCIL);
    }
```

To sum up, the security council has significant control over the current fault-proof system. If an attacker compromises the security council, there is a risk of theft of assets in the bridge.

Impact

Informational

Recommendation

In the short term, the security council should consist of sufficient independent entities with high reputations so that an attacker cannot easily compromise the security council.

In the long term, improvements are needed so that on-chain fault-proof works normally without a security council.

Patch

WIP

Until the long-term fix is implemented, the improved security council operation procedure (including secure communication and validation requirements for any security-critical operations) will migrate the risk.

#22 `KROMA-022` `KromaPortal.GUARDIAN` should be a MultiSig wallet

ID	Summary	Severity
`KROMA-022`	`KromaPortal.GUARDIAN` must be a multisig wallet due to its important permissions.	Informational

Description

In the testnet, the `GUARDIAN` address, which holds the pause permission for the `KromaPortal` contract is an EOA. If the private key for the `GUARDIAN` address is compromised, L2 users' funds may be frozen until the permission is revoked and unpaused.

Impact

Informational

Recommendation

A multisig wallet should be used as `GUARDIAN` instead of an EOA.

Patch

Fixed

The `GUARDIAN` will be the security council's multisig address.

#23 `KROMA-023` MultiSig wallet should confirm the transaction's content with its ID

ID	Summary	Severity
`KROMA-023`	The `MultiSigWallet` used by the security council may be problematic in the event of a reorg because it only uses the MultiSig transaction ID when confirming a MultiSig transaction.	Low

Description

`MultiSigWallet` executes the transaction when it receives sufficient owners' confirmation after a MultiSig transaction is created. However, because the MultiSig transaction to be confirmed is designated by only using ID, a different MultiSig transaction with the same ID but different content may be confirmed when a reorg occurs.

To exploit this vulnerability, an attacker would need to compromise the private key of a security council member and wait for an opportunity. When the security council is called, and the required confirmation transactions occur but are nullified by a reorg, along with the transaction for creating the MultiSig transaction, the attacker can execute the arbitrary MultiSig transaction. This can be achieved by creating their own MultiSig transaction, occupying the ID of a MultiSig transaction confirmed by security council members, and resubmitting it with the confirmation transactions.

Impact

Low

It requires the attacker to compromise the private key of a security council member, and the reorg must happen in favor of the attacker when the security council is called. Additionally, attempting this attack multiple times to increase chances is costly and likely to be detected.

However, if successful, it could allow the exploitation of the security council's permissions to inject a crafted L2 output root and steal the bridge's assets.

Recommendation

The `confirmTransaction` function should receive the hash of the transaction's detail to verify if it matches the transaction with the corresponding ID.

Patch

WIP

This issue will be addressed in the future, considering the requirement for a reorg and the compromise of the private key.

#24 `KROMA-024` Required bond amount for challengers should be limited

ID	Summary	Severity
`KROMA-024`	There should be an upper limit on the amount of bond required for the challenge to ensure its smooth progression.	Medium

Description

When the asserter submits the L2 output root, they can freely set a bond amount higher than the `MIN_BOND_AMOUNT`, and the same amount is required for the challenger. If there is a failed challenge, the bond amount is increased. If the bond required for the challenger becomes too large, or if the attacker creates a large bond from the beginning, the `CreateChallenge` transaction will fail due to insufficient funds, and the challenger will continuously retry the transaction, wasting gas.

Impact

Medium

If an invalid L2 output root is finalized due to failure to set a challenge, it can lead to theft of funds. However, if the [KROMA-015] issue is resolved, it will be difficult for the attacker to attempt such an attack because there is no guarantee that the bond amount can be recovered from their point of view since the protocol burns some amount of bond.

Recommendation

It is recommended to set the bond amount required for the challenger less than the minimum amount required for the asserter. However, it must be greater than the gas that the asserter must use to respond to the challenge.

Patch

Fixed

A revision was made to ensure that both the asserter and challenger will use a pre-set fixed bond amount.

#25 `KROMA-025` `createChallenge()` should have a check for L1 block hash to prevent challenger's loss due to L1 reorg

ID	Summary	Severity
`KROMA-025`	When creating a challenge, the challenge target is designated by using only the L2 output root index, thus if L1 reorg occurs, the challenger may incur losses.	Informational

Description

If L2 reorg occurs due to the L1 reorg, the validator (asserter) will resend a new L2 output root, and the previously sent transaction will not be processed because the L1 block hash does not match. However, when creating a challenge, only the L2 output root index is used to specify the target, and this can be a problem when L1 reorg occurs.

For example, if L1 reorg occurs, causing a change in the L2 output root, the validator will send a new L2 output root. When the challenger checks this event, if the challenger's rollup node has not yet processed L2 reorg caused by L1 reorg, it will be presumed that an incorrect output root has been submitted, and the challenge will be started.

Impact

Informational

Because honest challengers can incur losses, this can lead to low challenger participation, which can cause problems in operating the challenge system. However, the conditions for this to happen are difficult to meet.

Recommendation

For `Colosseum.createChallenge`, such as `L2OutputOracle.submitL2Output`, it is recommended to either handle reorg by receiving the L1 block hash information or receive data only from the finalized block.

Patch

Fixed

The first of the two recommendations was applied.

Revision History

Version	Date	Description
1.0	Sep 2, 2023	Initial version of report

Theori, Inc. (“We”) is acting solely for the client and is not responsible to any other party. Deliverables are valid for and should be used solely in connection with the purpose for which they were prepared as set out in our engagement agreement. You should not refer to or use our name or advice for any other purpose. The information (where appropriate) has not been verified. No representation or warranty is given as to accuracy, completeness or correctness of information in the Deliverables, any document, or any other information made available. Deliverables are for the internal use of the client and may not be used or relied upon by any person or entity other than the client. Deliverables are confidential and are not to be provided, without our authorization (preferably written), to entities or representatives of entities (including employees) that are not the client, including affiliates or representatives of affiliates of the client.

ChainLight