Kroma zkTrie Security Audit

: New zkTrie implementation for Kroma

April 5, 2024

Revision 1.21

ChainLight@Theori

Table of Contents

Executive Summary

Starting on Feb 11, 2024, ChainLight of Theori audited the new implementation of Kroma's zkTrie for the Kroma blockchain node software for a week. The implementation replaces the mirror of Scroll's zkTrie module, and is designed to integrate better with go-ethereum code while also being more performant for batch updates.

During our review, ChainLight found no security issues with the implementation, but did identify a panic reachable via the `debug` RPC namespace.

Audit Overview

Scope

Name	Kroma zkTrie Security Audit
Target / Version	• Git Repository (kroma-network/go-ethereum): commit ranges 0379233b1c5ea87444a79ea3170a06d811b4da0a ... 442e9a1edd3b7ff5d465a0aeca9d1920cb5a332f
Application Type	Blockchain node (L2)
Lang. / Platforms	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.
Note	Neutral information about the target that is not directly related to the project's safety and security.

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.
Patched	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	0	• N/A
High	1	• `ZKTRIE-003`
Medium	0	• N/A
Low	4	• `ZKTRIE-001` • `ZKTRIE-004` • `ZKTRIE-005` • `ZKTRIE-006`
Informational	1	• `ZKTRIE-002`
Note	0	• N/A

Findings

Summary

#	ID	Title	Severity	Status
1	ZKTRIE-001	`merkleTreeIterator.seek()` can panic due to key vs. path confusion	Low	Patched
2	ZKTRIE-002	Inconsistent Handling of unexpected HashNode	Informational	Patched
3	ZKTRIE-003	`MerkleTree.Delete` can incorrectly update the root node if removing a leaf at level 1.	High	Patched
4	ZKTRIE-004	Shallow copy can miscalculate the state root hash	Low	Patched
5	ZKTRIE-005	Key pre-image is not saved	Low	Patched
6	ZKTRIE-006	Invalid keyPreimage format	Low	Patched

#1 `ZKTRIE-001` `merkleTreeIterator.seek()` can panic due to key vs. path confusion

ID	Summary	Severity
`ZKTRIE-001`	`merkleTreeIterator.seek()` treats the input key as a path, leading to a possible `slice bounds out of range` error.	Low

Description

`merkleTreeIterator`s are created when the `NodeIterator()` method is called on a `ZkMerkleTrie`. As in the standard go-ethereum `Trie`, this method accepts a starting key for iteration. In the normal trie iterator (`nodeIterator`), this input key is transformed into nibbles (the equivalent of zkTrie paths):

```
func (it *nodeIterator) seek(prefix []byte) error {
    // The path we're looking for is the hex encoded key without terminato
r.
    key := keybytesToHex(prefix)
    key = key[:len(key)-1]
    ...
}
```

However in `merkleTreeIterator`, `seek()` assumes the input byte array is already in path form, leading to incorrect behavior and a possible panic.

```
func (it *merkleTreeIterator) seek(path []byte) {
    if len(path) == 0 {
        return
    }

    for _, p := range path {
        if parent, ok := it.stack[len(it.stack)-1].(*merkleTreeIteratorPar
entNode); ok {
```

```
            // AUDIT: this path is not validated to be valid, can cause OO
B access crash
            if child := it.resolveNode(parent.children[p]); child != nil {
                it.stack = append(it.stack, child)
                it.path = append(it.path, p)
                continue
            }
            ...
        }
        ...
    }
    ...
}
```

In most cases, the `start` key values are `nil`, so this issue is avoided. However, a non-`nil` start key can be passed via a go-ethereum dump command or by the `debug_accountRange` RPC method.

Impact

Low

Although the code is reachable by an RPC endpoint, the panic is caught and handled by the RPC handler.

Recommendation

Transform the start key into a path before usage.

```
diff --git a/trie/iterator.go b/trie/iterator.go
index c5198b741..921298ef9 100644
--- a/trie/iterator.go
+++ b/trie/iterator.go
@@ -929,13 +929,16 @@ func newMerkleTreeIterator(
        return it
 }

-func (it *merkleTreeIterator) seek(path []byte) {
+func (it *merkleTreeIterator) seek(start []byte) {
+       path := zk.NewTreePathFromBytes(start)
+
        if len(path) == 0 {
                return
        }
```

Remediation

Patched

It was already fixed in out-of-scope commits in a way similar to the recommendation.

#2 ZKTRIE-002 Inconsistent Handling of unexpected HashNode

ID	Summary	Severity
ZKTRIE-002	In some cases, encountering a `HashNode` produces the same result as an `EmptyNode`, when a new error type is warranted.	Informational

Description

In most `ZkMerkleTree` operations, encountering a `HashNode` yields a new type of error. However, in both `Delete()` and `Prove()`, the behavior instead matches that of an `EmptyNode`:

```
func (t *MerkleTree) Prove(key []byte, writeNode func(TreeNode) error) err
or {
    ....
        case *EmptyNode:
            return nil
        case *HashNode:
            return nil
    ....
}
```

```
func (t *MerkleTree) Delete(key []byte) error {
    ....
        case *EmptyNode:
            return trie.ErrKeyNotFound
        case *HashNode:
            return trie.ErrKeyNotFound
    ....
```

In both of these cases, encountering a `HashNode` should yield a new type of error.

Impact

Informational

If the implementation is correct, `HashNode`s should not be encountered. However, if a bug arises in the trie, these cases could hide the error and introduce incorrect outputs.

Recommendation

Return new error types, as is done in the other tree operations:

```
diff --git a/trie/zk/merkle_tree.go b/trie/zk/merkle_tree.go
index b7fed242f..3ea84c98c 100644
--- a/trie/zk/merkle_tree.go
+++ b/trie/zk/merkle_tree.go
@@ -247,7 +247,7 @@ func (t *MerkleTree) MustDelete(key []byte) {
 // mt.ImportDumpedLeafs), but this will lose all the Root history of the
MerkleTree
 func (t *MerkleTree) Delete(key []byte) error {
        node, path, pathNodes := t.rootNode, t.newTreePath(key), *new([]*P
arentNode)
-       for _, p := range path {
+       for lvl, p := range path {
                switch n := node.(type) {
                case *ParentNode:
                        pathNodes = append(pathNodes, n)
@@ -261,7 +261,7 @@ func (t *MerkleTree) Delete(key []byte) error {
                case *EmptyNode:
                        return trie.ErrKeyNotFound
                case *HashNode:
-                       return trie.ErrKeyNotFound
+                       return fmt.Errorf("Delete: encounter hash node. le
vel %d, path %v", lvl, path[:lvl])
                default:
                        return trie.ErrInvalidNodeFound
                }
@@ -336,7 +336,8 @@ func (t *MerkleTree) Prove(key []byte, writeNode func(
TreeNode) error) error {
                return err
        }
        node := t.rootNode
-       for _, p := range t.newTreePath(key) {
+       path := t.newTreePath(key)
+       for lvl, p := range path {
                // TODO: notice here we may have broken some implicit on t
he proofDb:
```

```
                // the key is not keccak(value) and it even can not be der
ived from the value by any means without an actual decoding
                if err := writeNode(node); err != nil {
@@ -350,7 +351,7 @@ func (t *MerkleTree) Prove(key []byte, writeNode func(
TreeNode) error) error {
                case *EmptyNode:
                        return nil
                case *HashNode:
-                       return nil
+                       return fmt.Errorf("Prove: encounter hash node. lev
el %d, path %v", lvl, path[:lvl])
                default:
                        return trie.ErrInvalidNodeFound
                }
```

Remediation

Patched

It is fixed as recommended.

#3 `ZKTRIE-003` `MerkleTree.Delete` can incorrectly update the root node if removing a leaf at level 1.

ID	Summary	Severity
`ZKTRIE-003`	`MerkleTree.Delete` can incorrectly update the root node if removing a leaf at level 1.	High

Description

`MerkleTree.Delete` can incorrectly update the root node if removing a leaf at level 1. If a `LeafNode` at level 1 is deleted, its sibling is being promoted to the root node. This behavior is incorrect when the sibling is a `ParentNode`, as it changes the path prefix of all nodes below the promoted `ParentNode`. Instead, the deleted `LeafNode` should be replaced by an `EmptyNode`.

Impact

High

1. Some value on the state db can be removed.
2. So that it can miscalculate the state root hash, and it leads to the fork.

Recommendation

```
diff --git a/trie/zk/merkle_tree.go b/trie/zk/merkle_tree.go
index 3ea84c98c..5c3cbd38b 100644
--- a/trie/zk/merkle_tree.go
+++ b/trie/zk/merkle_tree.go
@@ -275,10 +275,6 @@ func (t *MerkleTree) rmAndUpload(path TreePath, pathN
odes []*ParentNode) {
        switch len(pathNodes) {
        case 0: // The leaf node you want to remove is root node.
                t.rootNode = EmptyNodeValue
-       case 1:
-               // root (ParentNode) --- LeafNode or ParentNode (promoted
to root node)
-               //                    |- LeafNode (deleted)
-               t.rootNode = t.getChild(pathNodes[0], path.GetOther(0))
        default:
                lastSibling := t.getChild(pathNodes[len(pathNodes)-1], pat
h.GetOther(len(pathNodes)-1))
```

Remediation

Patched

It is patched as recommended.

#4 `ZKTRIE-004` Shallow copy can miscalculate the state root hash

ID	Summary	Severity
`ZKTRIE-004`	Shallow copy can miscalculate the state root hash.	Low

Description

Any time a node is mutated in a way which could change its hash (i.e. SetChild), first copy it. When SetChild is only being used to replace a HashNode with its real node, the hash will not change, so we do not need to copy as long as we don't accidentally clear the hash. Multiple threads could be doing the replacement concurrently, so we need to be more careful about detecting this case. This is handled by comparing the child hashes instead of checking the node type.

Impact

Low

The statedb calculation could failed.

Recommendation

```
diff --git a/trie/zk/merkle_tree.go b/trie/zk/merkle_tree.go
index 3c7492bbd..40d9d1abb 100644
--- a/trie/zk/merkle_tree.go
+++ b/trie/zk/merkle_tree.go
@@ -182,6 +182,7 @@ func (t *MerkleTree) addLeaf(
                        log.Error("fail to addLeaf", "err", err, "level",
lvl)
                        return nil, err
                }
+               n = n.Copy()
                n.SetChild(path.Get(lvl), newNode) // Update the node to r
eflect the modified child
                return n, nil
        case *LeafNode:
@@ -250,6 +251,10 @@ func (t *MerkleTree) Delete(key []byte) error {
        for lvl, p := range path {
                switch n := node.(type) {
                case *ParentNode:
+                       n = n.Copy()
+                       if lvl > 0 {
+                               pathNodes[len(pathNodes)-1].SetChild(path.
Get(lvl-1), n)
+                       }
                        pathNodes = append(pathNodes, n)
                        node = t.getChild(n, p)
                case *LeafNode:
diff --git a/trie/zk/merkle_tree_node.go b/trie/zk/merkle_tree_node.go
index b89214d56..522345807 100644
--- a/trie/zk/merkle_tree_node.go
+++ b/trie/zk/merkle_tree_node.go
@@ -68,6 +68,10 @@ func newParentNodeFromBlob(blob []byte) (*ParentNode, e
rror) {
        }, nil
 }

+func (n *ParentNode) Copy() *ParentNode {
+       return &ParentNode{childL: n.childL, childR: n.childR, hash: n.has
```

```
h}
+}
+
 func (n *ParentNode) Hash() *zkt.Hash { return n.hash }

 func (n *ParentNode) CanonicalValue() []byte {
@@ -92,8 +96,8 @@ func (n *ParentNode) SetChild(path byte, child TreeNode)
 {
        } else {
                n.childL = child
        }
-       if _, ok := oldChild.(*HashNode); ok && child.Hash() != nil && byt
es.Equal(oldChild.Hash()[:], child.Hash()[:]) {
-               // This is a case of converting a HashNode to the original
 TreeNode. Does not clear the hash.
+       if oldChild.Hash() != nil && child.Hash() != nil && bytes.Equal(ol
dChild.Hash()[:], child.Hash()[:]) {
+               // The child hash has not changed. Does not clear the hash
.
                return
        }
        n.hash = nil
```

Remediation

Patched

#5 `ZKTRIE-005` Key pre-image is not saved

ID	Summary	Severity
`ZKTRIE-005`	Key pre-image is not stored during translation, so can not be fetched during proof generation.	Low

Description

In `newZKMerkleStateTrie`, `transformKey` is set to a function which translates the key to its secure hash:

```
func newZkMerkleStateTrie(tree *zk.MerkleTree, db *Database) *ZkMerkleStat
eTrie {
    trie := NewZkMerkleTrie(tree, db)
    trie.logger = log.New("trie", "ZkMerkleStateTrie")
    trie.transformKey = func(key []byte) ([]byte, error) {
        sanityCheckByte32Key(key)
        hash, err := zk.NewSecureHash(key)
        if err != nil {
            return nil, err
        }
        return hash[:], nil
    }
    return &ZkMerkleStateTrie{ZkMerkleTrie: trie, preimage: db.preimages}
}
```

However, the key preimage is not stored in `db.preimages`, so the `trie.GetKey` will fail to lookup the key:

```
func (z *ZkMerkleStateTrie) GetKey(kHashBytes []byte) []byte {
    // TODO: use a kv cache in memory
    k, err := zkt.NewBigIntFromHashBytes(kHashBytes)
    if err != nil {
        z.logger.Error("failed to GetKey", "error", err)
        return nil
```

```
    }
    if z.db.preimages == nil {
        return nil
    }
    return z.db.preimages.preimage(common.BytesToHash(k.Bytes()))
}
```

Impact

Low

This issue would only impact proof generation, causing temporary downtime until the issue is resolved.

Recommendation

Store the key pre-image in `db.preimage` during key transformation.

Remediation

Patched

#6 `ZKTRIE-006` Invalid keyPreimage format

ID	Summary	Severity
`ZKTRIE-006`	The format of `keyPreimage` may be invalid in encoded leaf nodes for proofs.	Low

Description

In `ZkMerkleStateTrie.Prove`, a callback is passed to `ZkMerkleTrie.prove` which encodes each proof node and adds to to the proof DB:

```
func (z *ZkMerkleStateTrie) Prove(key []byte, proofDb ethdb.KeyValueWriter
) error {
    return z.prove(common.ReverseBytes(key), proofDb, func(node zk.TreeNod
e) error {
        value := node.CanonicalValue()
        if leaf, ok := node.(*zk.LeafNode); ok {
            if preImage := z.GetKey(common.ReverseBytes(leaf.Key)); len(pr
eImage) > 0 {
                value[len(value)-1] = byte(len(preImage))
                value = append(value, preImage[:]...)
            }
        }
        return proofDb.Put(node.Hash()[:], value)
    })
}
```

When encountering a `LeafNode`, it correctly attempts to add the `keyPreimage` to the encoded node, which is required for proof verification. However, with this encoding, proof verification on-chain will only work if the `keyPreimage` length is 32 bytes. As a result, shorter preimages may fail to verify on-chain.

Impact

Low

This issue would only impact proof generation, causing temporary downtime until the issue is resolved.

Recommendation

Canonicalize the preimage to 32 bytes before appending, making it compatible with the verification contracts.

Remediation

Patched

Revision History

Version	Date	Description
1.0	Feb 23, 2024	Initial version
1.1	Mar 15, 2024	Update ZKTRIE-003, 004
1.2	April 5, 2024	Update ZKTRIE-005, 006
1.21	April 5, 2024	Revised remediation status