Kroma zkTrie Security Audit

: New zkTrie implementation for Kroma

Feb 23, 2024

Revision 1.0

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.

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	0	• N/A
High	0	• N/A
Medium	0	• N/A
Low	1	• `ZKTRIE-001`
Informational	1	• `ZKTRIE-002`

Findings

Summary

#	ID	Title	Severity	Status
1	`ZKTRIE-001`	`merkleTreeIterator.seek()` can panic due to key vs. path confusion	Low	Fixed
2	`ZKTRIE-002`	Inconsistent Handling of unexpected HashNode	Informational	Fixed

#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
        }
```

Patch

Fixed

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
                }
```

Patch

Fixed

It is fixed as recommended.

Revision History

Version	Date	Description
1.0	Feb 23, 2024	Initial version

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