SHERLOCK

Security Review For MegaETH

Collaborative Audit Prepared For: MegaETH
Lead Security Expert(s): blockdev
Koolex
oniki
sammy
Date Audited: August 26 - September 16, 2025

Introduction

SALT (Small Authentication Large Trie) is a novel authenticated key-value store that powers MegaETH. SALT is highly memory- and I/O-efficient. For up to 3 billion key-value pairs, SALT's authentication layer requires only a 1 GB memory footprint, and it scales smoothly to tens of billions of items. Thus, SALT can fit entirely in the memory of most modern machines, relieving blockchain nodes of expensive random disk I/Os. To the best of our knowledge, SALT is the first authenticated KV store to scale to tens of billions of items and completely eliminate random disk I/Os during state root updates, all while maintaining its low memory footprint.

Scope

Repository: megaeth-labs/salt

Audited Commit: 5cfa144985ec15b8721a652620de0569ed34d57d

Final Commit: ad72d82b67f119613cf3cf4491223c7b159eb66e

Files:

- banderwagon/Cargo.toml
- banderwagon/docs/1-understand-banderwagon-high-level.md
- banderwagon/docs/2-understand-banderwagon-twisted-edwards.md
- banderwagon/docs/3-understand-banderwagon-point-halving.md
- banderwagon/readme.md
- banderwagon/src/element.rs
- banderwagon/src/lib.rs
- banderwagon/src/msm.rs
- banderwagon/src/salt_committer.rs
- banderwagon/src/scalar_multi_asm.rs
- banderwagon/src/trait_impls/ops.rs
- banderwagon/src/trait_impls.rs
- banderwagon/src/trait_impls/serialize.rs
- ipa-multipoint/benches/benchmark_main.rs
- ipa-multipoint/benches/benchmarks/ipa_prove.rs
- ipa-multipoint/benches/benchmarks/ipa_verify.rs
- ipa-multipoint/benches/benchmarks/mod.rs
- ipa-multipoint/benches/benchmarks/multipoint_prove.rs

- ipa-multipoint/benches/benchmarks/multipoint_verify.rs
- ipa-multipoint/Cargo.toml
- ipa-multipoint/docs/1-vcs-high-level.md
- ipa-multipoint/docs/2-vcs-multipoint-arg.md
- ipa-multipoint/docs/3-vcs-divide-lagrange-basis.md
- ipa-multipoint/LICENSE
- ipa-multipoint/Readme.md
- ipa-multipoint/src/crs.rs
- ipa-multipoint/src/default_crs.rs
- ipa-multipoint/src/ipa.rs
- ipa-multipoint/src/lagrange_basis.rs
- ipa-multipoint/src/lib.rs
- ipa-multipoint/src/main.rs
- ipa-multipoint/src/math_utils.rs
- ipa-multipoint/src/multiproof.rs
- ipa-multipoint/src/transcript.rs
- README.md
- salt/benches/salt_trie.rs
- salt/Cargo.toml
- salt/README.md
- salt/src/constant.rs
- salt/src/empty_salt.rs
- salt/src/lib.rs
- salt/src/mem_store.rs
- salt/src/mock_evm_types.rs
- salt/src/state/hasher.rs
- salt/src/state/mod.rs
- salt/src/state/state.rs
- salt/src/state/updates.rs
- salt/src/traits.rs
- salt/src/trie/mod.rs

- salt/src/trie/node_utils.rs
- salt/src/trie/trie.rs
- salt/src/types.rs

Final Commit Hash

ad72d82b67f119613cf3cf4491223c7b159eb66e

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
5	3	7

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Use of panics, expects, asserts without error handling [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/63

This issue has been acknowledged by the team but won't be fixed at this time.

Description

The SALT codebase contains numerous instances of `panic!`, `assert!`, `.expect()` calls that should be handled gracefully with proper error handling. These can cause node crashes.

Non exhaustive list of problematic panics:

1. State Management (`state/state.rs`)

- **Line 232:** `v.try_into().expect("Failed to decode bucket metadata")` - Panics on invalid metadata
- **Line 464-467:** `assert!(new_capacity <= BUCKET_SLOT_ID_MASK)` - Panics on oversized capacity

2. Trie Operations (`trie/trie.rs`)

- **Line 286**: `.expect("old meta exist in updates")` - Panics if metadata missing
- **Line 288**: `.expect("old meta should be valid")` - Panics on invalid metadata conversion
- **Line 292**: `.expect("new meta exist in updates")` - Panics if new metadata missing
- **Line 294**: `.expect("new meta should be valid")` - Panics on invalid conversion
- **Line 403**: `.expect("node should exist in trie")` - Panics if node missing
- **Line 485**: `.expect("root node should exist in trie")` - Panics if root missing
- **Line 496**: `.expect("root node should exist in trie")` - Panics if root missing
- **Line 545**: `.expect("update internal nodes for subtrie failed")` - Panics on internal node failure

3. Type System (`types.rs`)

- **Line 364**: `panic!("metadata buckets cannot have subtries")` - Panics on invalid bucket type
- **Line 32**: `.expect("Failed to serialize scalar to bytes")` - Panics on serialization failure

4. Proof System (`proof/prover.rs`)

- **Line 514**: `.expect("Failed to serialize scalar to bytes")` - Panics on serialization failure

5. Witness System (`proof/salt_witness.rs`)

- **Line 591**: `panic!("Expected Ok(Some(_)), got {:?}", other)` - Panics on witness validation
- **Line 597**: `panic!("Expected Ok(None), got {:?}", other)` - Panics on witness validation
- **Line 603**: `panic!("SECURITY VIOLATION: Unknown key returned Ok(None)")` - Panics on security violation
- **Line 604**: `panic!("SECURITY VIOLATION: Unknown key returned Ok(Some(_))")` - Panics on security violation

Fix

Replace all panic-causing operations with proper error handling

Discussion

yilongli

We will go through all these panic-causing operations. Perhaps some of them can be changed into error propagation. However, panic is indeed the right decision in many of the cases because there is no proper error handling.

lpetroulakis

The issue is acknowledged by the team, but there is no need for a fix at this time.

Issue H-2: shi_delete unwrap panic and cache consistency bugs in SALT state management [RESOLVED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/64

Summary

Multiple critical bugs exist in the SALT state management related to bucket usage count handling and cache consistency, causing panics during stateless validation and cache inconsistencies between `bucket_used_cache` and the internal cache.

Vulnerability Detail

1. `shi_delete` Panics Without Bucket Usage Count

The `shi_delete` method unconditionally calls `unwrap()` on `metadata.used` without checking if it's `None`:

```
fn shi_delete(
    &mut self,
    bucket_id: BucketId,
    key: &[u8],
    out_updates: &mut StateUpdates,
) -> Result<(), Store::Error> {
    let metadata = self.metadata(bucket_id, true)?;
    if let Some((slot, salt_val)) =
        self.shi_find(bucket_id, metadata.nonce, metadata.capacity, key)?
    {
        // Update the bucket usage cache
        self.bucket_used_cache
            .insert(bucket_id, metadata.used.unwrap() - 1);  // <-- PANIC HERE
```

https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/blob/d5a3cad2f9d4ac796a46927899385f000441d8a7/salt/salt/src/state/state.rs#L412-L414

When using `SaltWitness` as the storage backend (during stateless validation), the `metadata.used` field is intentionally set to `None` to optimize witness size. This causes a panic on deletion.

Upon inspecting the test case provided for this first bug, the dev team discovered three additional related bugs:

2. Cache Inconsistency Issues

The `EphemeralSaltState::bucket_used_cache` should be consistent with `EphemeralSaltState::cache` at all times - whenever inserting/deleting a key-value pair in cache, `bucket_used_cache` should be updated accordingly. This invariant is broken in multiple places:

- `shi_upsert` breaks this invariant in both old and patched implementations
- `shi_delete` breaks this invariant in patched implementations

3. Unnecessary Usage Count Requirement for Updates

`shi_upsert` incorrectly requires bucket usage count even when just updating an existing key-value pair (not inserting new). This is inefficient as updates don't change the bucket's used slot count.

4. Confusing `metadata` Behavior

The `metadata(..., need_used)` method has unclear semantics. According to the dev team:

- Current behavior: `need_used = true` means "do your best to get the bucket usage count but don't generate an explicit error upon failure"
- Expected invariant: If `need_used` is `true`, the returned `BucketMeta.used` field should be guaranteed to be `Some(_)`. If `false`, it should be `None`.

This confusing behavior is error-prone, yet `shi_upsert` depends on it to pass tests.

Impact

As noted by the dev team: "This is indeed a critical bug because now stateless validation could come to a different conclusion (i.e., panic) than the sequencer even with a totally valid witness."

The bugs cause:

- Application panics during stateless validation when deleting keys
- Cache inconsistencies that could lead to incorrect state computations
- Unnecessary performance overhead when updating existing keys

Code Snippet

Test case demonstrating the `shi_delete` panic:

```
#[test]
fn test_shi_delete_without_usage_count() {
    use crate::proof::salt_witness::create_witness;

    // Create a witness with an existing key and metadata with used: None
    let existing_key = [1u8; 32];
    let hashed = hasher::hash_with_nonce(&existing_key, 0);
    let slot = probe(hashed, 0, 4);

    let witness = create_witness(
        TEST_BUCKET,
        Some(BucketMeta {
            nonce: 0,
            capacity: 4,
            used: None, // This will cause the panic
        }),
        vec![(slot, Some(SaltValue::new(&existing_key, &[2u8; 32])))],
    );

    let mut state = EphemeralSaltState::new(&witness);
    let mut updates = StateUpdates::default();

    // This WILL panic with "called `Option::unwrap()` on a `None` value"
    state
        .shi_delete(TEST_BUCKET, &existing_key, &mut updates)
        .unwrap();
}
```

Tool Used

Manual Review

Recommendation

Notes and patch as per the dev team

> The insight is that the following two things are fundamentally in conflict:
> 1. update bucket_used_cache for every insert/delete
> 2. shi_delete must succeed without knowing the bucket usage count The solution is to track the net changes of bucket usage counts instead of absolute values. The rest of the problems are easy fix.

The patch

```
diff --git a/salt/src/state/state.rs b/salt/src/state/state.rs
index b67bb75..011459e 100644
--- a/salt/src/state/state.rs
+++ b/salt/src/state/state.rs
@@ -84,14 +84,17 @@ pub struct EphemeralSaltState<'a, Store> {
     /// Note: This field is `pub(crate)` to enable proof generation modules to
     /// access the set of touched keys for witness construction.
     pub(crate) cache: HashMap<SaltKey, Option<SaltValue>>,
-    /// Tracks the current usage count for buckets modified in this session.
+    /// Tracks the net change in bucket usage counts relative to the base store.
+    ///
+    /// Each value represents the delta from the base store's bucket usage count:
+    /// - +1 for each insertion, -1 for each deletion
+    /// - Missing entries: no net change from base state
     ///
     /// This field is essential because when [`BucketMeta`] is serialized to
      ↪  [`SaltValue`],
     /// only the `nonce` and `capacity` fields are preserved - the usage count is
      ↪  dropped.
-    /// Without this cache, computing the current bucket occupancy would require
 ↪  complex
-    /// logic to reconcile the base store's usage count with all insertions and
 ↪  deletions
-    /// tracked in the main cache.
-    bucket_used_cache: HashMap<BucketId, u64>,
+    /// Without this delta tracking, computing the current bucket occupancy would
 ↪  require
+    /// reconciling the base store's usage count with all cached modifications.
+    usage_count_delta: HashMap<BucketId, i64>,
     /// Whether to cache values read from the store for subsequent access
     cache_read: bool,
 }
@@ -105,7 +108,7 @@ impl<'a, Store: StateReader> EphemeralSaltState<'a, Store> {
         Self {
             store,
             cache: HashMap::new(),
-            bucket_used_cache: HashMap::new(),
+            usage_count_delta: HashMap::new(),
            cache_read: false,
         }
     }
@@ -226,6 +229,25 @@ impl<'a, Store: StateReader> EphemeralSaltState<'a, Store> {
         Ok(state_updates)
     }

+    /// Try our best to get the number of used slots in a bucket.
+    ///
+    /// This method computes the current usage by adding any tracked deltas from
+    /// `usage_count_delta` to the base count from the underlying store.
+    ///
+    /// # Arguments
+    /// * `bucket_id` - The bucket ID to get the usage count for
+    ///
+    /// # Returns
+    /// * `Ok(count)` - The number of used slots in the bucket
+    /// * `Err(error)` - If the store query fails
+    fn try_bucket_used_slots(&self, bucket_id: BucketId) -> Result<u64,
↪  Store::Error> {
+        let base_count = self.store.bucket_used_slots(bucket_id)?;
+        let result = base_count as i64 +
↪  self.usage_count_delta.get(&bucket_id).unwrap_or(&0);
+        Ok(result
+            .try_into()
+            .expect("Bucket usage count became negative"))
+    }
+
     /// Retrieves bucket metadata for the given bucket ID.
     ///
     /// # Arguments
@@ -234,6 +256,10 @@ impl<'a, Store: StateReader> EphemeralSaltState<'a, Store> {
     ///   avoids unnecessary `bucket_used_slots()` calls to the underlying storage
     ///   backend when the usage count is not needed (e.g., for read operations
      ↪  like
     ///   `plain_value` that only require `nonce` and `capacity`).
+    ///
+    /// # Invariant
+    /// If `need_used` is `true`, the returned `BucketMeta.used` field is
↪  guaranteed
+    /// to be `Some(_)`. If `need_used` is `false`, the `used` field will be
↪  `None`.
     fn metadata(
         &mut self,
         bucket_id: BucketId,
@@ -241,45 +267,19 @@ impl<'a, Store: StateReader> EphemeralSaltState<'a, Store> {
     ) -> Result<BucketMeta, Store::Error> {
         let metadata_key = bucket_metadata_key(bucket_id);

-        // Check cache first
-        let meta = if let Some(cached_value) = self.cache.get(&metadata_key) {
+        // Get nonce + capacity (either from cache or store)
+        let mut meta = if let Some(cached_value) = self.cache.get(&metadata_key) {
             // Found in cache - decode it
-            let mut meta = match cached_value {
+            match cached_value {
                 Some(v) => v.try_into().expect("Failed to decode bucket metadata"),
                 None => BucketMeta::default(),
-            };
-
-            // Cached value doesn't have 'used' field, populate if needed
-            if need_used {
-                meta.used = Some(
-                    if let Some(&used) = self.bucket_used_cache.get(&bucket_id) {
-                        used
-                    } else {
-                        self.store.bucket_used_slots(bucket_id)?
-                   },
-               );
            }
-           meta
        } else {
            // Common path: Not in cache, get from store (more efficient than
            ↪  self.value())
-           let mut meta = self.store.metadata(bucket_id)?;
-
-           if need_used {
-               // Look up cache for the latest usage count
-               if let Some(&cached_used) = self.bucket_used_cache.get(&bucket_id)
↪  {
-                   meta.used = Some(cached_used);
-               }
-           } else {
-               // Clear "used" if not needed
-               meta.used = None;
-           }
+           let meta = self.store.metadata(bucket_id)?;

            // Cache the metadata only if requested
            if self.cache_read {
-               // Performance note: We intentionally avoid caching the usage count
-               // to save on HashMap insertions. Since plain keys are distributed
-               // randomly across buckets, bucket metadata is rarely reused
↪  between
-               // different keys.
                self.cache.insert(
                    metadata_key,
                    if meta.is_default() {
@@ -292,6 +292,14 @@ impl<'a, Store: StateReader> EphemeralSaltState<'a, Store> {
            meta
        };

+       // Get usage count only if requested
+       meta.used = if need_used {
+           Some(self.try_bucket_used_slots(bucket_id)?)
+       } else {
+           // Clear `used` if not needed (must not return a stale base count)
+           None
+       };
+
        Ok(meta)
    }

@@ -322,7 +330,7 @@ impl<'a, Store: StateReader> EphemeralSaltState<'a, Store> {
        }

        // Step 2: Fall through to standard SHI upsert algorithm
```

```
-        let metadata = self.metadata(bucket_id, true)?;
+        let metadata = self.metadata(bucket_id, false)?;
         let hashed_key = hasher::hash_with_nonce(plain_key, metadata.nonce);

         // Linear probe through the bucket, creating a displacement chain
@@ -358,11 +366,10 @@ impl<'a, Store: StateReader> EphemeralSaltState<'a, Store> {
                 // Found empty slot - insert the key and complete
                 self.update_value(out_updates, salt_key, None, Some(new_salt_val));

-                if let Some(mut used) = metadata.used {
-                    // Update the bucket usage cache.
-                    used += 1;
-                    self.bucket_used_cache.insert(bucket_id, used);
+                // Update the usage count delta for this insertion
+                *self.usage_count_delta.entry(bucket_id).or_insert(0) += 1;

+                if let Ok(used) = self.try_bucket_used_slots(bucket_id) {
                     // Resize the bucket if load factor threshold exceeded
                     if used > metadata.capacity * BUCKET_RESIZE_LOAD_FACTOR_PCT /
                     ↪  100 {
                         let new_capacity =
                          ↪  compute_resize_capacity(metadata.capacity, used);
@@ -386,7 +393,7 @@ impl<'a, Store: StateReader> EphemeralSaltState<'a, Store> {
                     // 2. It can only delay bucket resizing (temporary deviation
                     ↪  from the
                     //    canonical state)
                     // 3. The worst case: a malicious sequencer causes the bucket
                     ↪  to exceed
-                    //    its ideal capacity, degrading performance but not
↪  correctness
+                    //    its ideal load factor, degrading performance but not
↪  correctness
                     //
                     // ## Self-Healing Mechanism
                     // When a legitimate sequencer performs the next insertion, it
                     ↪  will:
@@ -421,13 +428,12 @@ impl<'a, Store: StateReader> EphemeralSaltState<'a, Store> {
         key: &[u8],
         out_updates: &mut StateUpdates,
     ) -> Result<(), Store::Error> {
-        let metadata = self.metadata(bucket_id, true)?;
+        let metadata = self.metadata(bucket_id, false)?;
         if let Some((slot, salt_val)) =
             self.shi_find(bucket_id, metadata.nonce, metadata.capacity, key)?
         {
-            // Update the bucket usage cache
-            self.bucket_used_cache
-                .insert(bucket_id, metadata.used.unwrap() - 1);
+            // Update the usage count delta for this deletion
+            *self.usage_count_delta.entry(bucket_id).or_insert(0) -= 1;
```

```
            // Slot compaction: retrace the displacement chain created by
             ↪  shi_upsert.
            // When shi_upsert displaced keys to make room, deletion must undo this
@@ -1150,8 +1156,8 @@ mod tests {
            "Cache should contain exactly 3 entries: bucket metadata + 2 key-value
             ↪  pairs"
        );
        assert!(
-            cache_state.bucket_used_cache.is_empty(),
-            "Bucket usage cache should remain empty - we don't cache bucket usage
↪  counts on read"
+            cache_state.usage_count_delta.is_empty(),
+            "Usage count delta should remain empty"
        );
    }

@@ -1404,8 +1410,7 @@ mod tests {
    /// - **Usage count behavior**: Validates `need_used=false` (no usage) vs
     ↪  `need_used=true` (populates usage)
    /// - **Stored metadata**: Tests custom metadata retrieval (nonce=123,
     ↪  capacity=512)
    /// - **Non-zero usage counting**: Adds data to bucket and verifies usage
     ↪  count reflects actual entries
-    /// - **Cache behavior**: Tests that `bucket_used_cache` takes precedence over
↪  store counting
-    /// - **Cache isolation**: Verifies usage counts aren't cached during read
↪  operations
+    /// - **Cache behavior**: Tests that `usage_count_delta` is used in usage
↪  calculations
    #[test]
    fn test_metadata() {
        let store = MemStore::new();
@@ -1449,18 +1454,15 @@ mod tests {
        assert_eq!(meta.nonce, 123);
        assert_eq!(meta.capacity, 512);
        assert_eq!(meta.used, Some(1)); // Now has 1 slot used
-        assert!(
-            state.bucket_used_cache.is_empty(),
-            "Usage counts not cached on reads"
-        );
+        assert!(state.usage_count_delta.is_empty());

-        // Test 5: Cached usage count (should use cache instead of calling
↪  bucket_used_slots)
-        state.bucket_used_cache.insert(TEST_BUCKET, 100);
+        // Test 5: Usage count delta (should use base count + delta)
+        state.usage_count_delta.insert(TEST_BUCKET, 99); // +99 delta on top of
↪  existing 1
```

```
        let meta = state.metadata(TEST_BUCKET, true).unwrap();
        assert_eq!(meta.nonce, 123);
        assert_eq!(meta.capacity, 512);
-       assert_eq!(meta.used, Some(100)); // From cache, not store count
+       assert_eq!(meta.used, Some(100)); // 1 (base) + 99 (delta) = 100
    }

    /// Test insertion order independence with small bucket and no resize.
@@ -1734,8 +1736,8 @@ mod tests {
        let retrieved = state.value(salt_key).unwrap().unwrap();
        assert_eq!(retrieved.value(), [2u8; 32].as_slice());

-       // Verify side effects: no cache update, no resize
-       assert!(state.bucket_used_cache.is_empty());
+       // Verify side effects: delta tracked (insertion happened), no resize
+       assert_eq!(state.usage_count_delta.get(&TEST_BUCKET), Some(&1));
        assert_eq!(state.metadata(TEST_BUCKET, false).unwrap().capacity, 4);
    }

@@ -1787,6 +1789,42 @@ mod tests {
        assert_eq!(retrieved.value(), [99u8; 32].as_slice());
    }

+   /// Tests shi_delete when the bucket usage count is unknown (incomplete
↪  witness).
+   #[test]
+   fn test_shi_delete_update_usage_count() {
+       use crate::proof::salt_witness::create_witness;
+
+       // Create a witness with two known slots: an existing key and an empty
+       // slot next to it
+       let existing_key = [1u8; 32];
+       let hashed = hasher::hash_with_nonce(&existing_key, 0);
+       let slot = probe(hashed, 0, 4);
+
+       let witness = create_witness(
+           TEST_BUCKET,
+           Some(BucketMeta {
+               nonce: 0,
+               capacity: 4,
+               used: None,
+           }),
+           vec![
+               (slot, Some(SaltValue::new(&existing_key, &[2u8; 32]))),
+               ((slot + 1) % 4, None),
+           ],
+       );
+
+       let mut state = EphemeralSaltState::new(&witness);
+       let mut updates = StateUpdates::default();
```

```
+
+        // shi_delete algorithm should complete successfully
+        assert!(state
+            .shi_delete(TEST_BUCKET, &existing_key, &mut updates)
+            .is_ok());
+
+        // Deletion should always create a -1 delta entry
+        assert_eq!(state.usage_count_delta.get(&TEST_BUCKET), Some(&-1));
+    }
+
     /// Comprehensive test for shi_rehash method covering various scenarios.
     #[test]
     fn test_shi_rehash() {
diff --git a/salt/src/traits.rs b/salt/src/traits.rs
index 62bf962..43c5e02 100644
--- a/salt/src/traits.rs
+++ b/salt/src/traits.rs
@@ -71,8 +71,10 @@ pub trait StateReader: Debug + Send + Sync {
     ///
     /// This method looks up the bucket's metadata using the bucket ID. If no
      ↪  metadata
     /// entry exists, it uses default values for `nonce` (0) and `capacity`
      ↪  (MIN_BUCKET_SIZE).
-    /// Regardless of whether metadata is stored or default, the `used` field is
 ↪  **always**
-    /// populated with the actual number of occupied slots.
+    /// The `used` field behavior is implementation-defined:
+    /// - Full node implementations may populate it with the actual number of
 ↪  occupied slots
+    /// - Witness implementations may leave it as `None` if they lack sufficient
 ↪  information
+    /// - For reliable slot counts, use the `bucket_used_slots()` method instead
     ///
     /// # Arguments
     ///
```

Issue H-3: Data loss in shi_rehash due to missing capacity validation during bucket downsizing [RESOLVED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/67

Summary

`shi_rehash` can cause data loss when downsizing a bucket without validating that the new capacity is sufficient to hold all existing entries.

Vulnerability Detail

The `shi_rehash` function in `state.rs` allows reducing bucket capacity without checking if all existing entries will fit in the new size. When rehashing with a smaller capacity than the number of existing entries, some key-value pairs cannot find slots during reinsertion and are silently lost.

The function extracts all existing entries, then attempts to reinsert them into the new bucket structure. However, if `new_capacity < number_of_entries`, the reinsertion loop in Step 3 will fail to place all entries, resulting in permanent data loss.

https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/blob/d5a3cad2f9d4ac796a46927899385f000441d8a7/salt/salt/src/state/state.rs#L469-L491

Test case demonstrating the bug:

```
#[test]
fn test_data_loss_by_shi_rehash() {
    // Test cases: (old_nonce, old_capacity, new_nonce, new_capacity, num_entries)
    let test_cases = [
        (0, 16, 0, 8, 9), // Capacity contraction
    ];

    // reducing from 16 to 8, will cause a data loss, which will make the test fail
    // because verify_all_keys_present is called in verify_rehash
    // change from 8 to 9, and the test will succeed

    for (old_nonce, old_capacity, new_nonce, new_capacity, num_entries) in
    ↪   test_cases {
        verify_rehash(
            old_nonce,
            old_capacity,
            new_nonce,
            new_capacity,
            num_entries,
        );
    }
}
```

Impact

Data loss occurs when bucket capacity is reduced below the number of stored entries. This breaks the fundamental guarantee of the SHI (Strongly History Independent) hash table that all key-value pairs should be preserved during rehashing operations.

Code Snippet

See vulnerability detail section.

Tool Used

Manual Review

Recommendation

Add validation after Step 2 to ensure the new capacity can accommodate all existing entries:

```
// After Step 2, before Step 3:
let entry_count = plain_kv_pairs.len();
if entry_count > new_capacity as usize {
    return Err(Store::Error::InsufficientCapacity);
}
```

Alternatively, automatically adjust the new capacity to the minimum required size:

```
let new_capacity = new_capacity.max(plain_kv_pairs.len() as u64);
```

Issue H-4: Overflow attack in probing [RESOLVED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/73

Summary

`step` plus `hashed_key` potentially overflows

Vulnerability Detail

`hashed_key` is 64 bit number and truncated result of hash of a `plain_key` It is feasible for an attacker to find a plain_key for example an ethereum address to hit `u64::MAX`. In that case probing will cause panic.

Impact

Attacker can halt prover and verifier.

Code Snippet

```
#[inline(always)]
pub(crate) fn probe(hashed_key: u64, i: u64, capacity: u64) -> SlotId {
    ((hashed_key + i) % capacity) as SlotId
}
```

Tool Used

Manual Review

Recommendation

Appling modular reduction in advance should resolve the issue

Issue H-5: Verifier fails with valid proof [RESOLVED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/74

Summary

This case is found while soft fuzzing salt API. Verifier failed in basic integration test when we force a bucket to expand with +250 same bucket keys

Vulnerability Detail

While creating verifier queries even `proof_node_ids`, `required_node_ids` are set equal `required_node_ids` is not sorted by key.

Impact

Verifier denies correct transition

Code Snippet

https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/blob/d5a3cad2f9d4ac796a46927899385f000441d8a7/salt/salt/src/proof/prover.rs#L251

Tool Used

Manual Review

Recommendation

Temporarily I've achieved to fix the issue by simply sorting `required_node_ids`

Issue M-1: Incorrect y coordinate selection logic in `get_point_from_x` function [RESOLVED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/61

Description

The `get_point_from_x` function has incorrect logic for selecting the y-coordinate when `choose_largest` is false. When `y` is already negative (`is_largest == false`) and `choose_largest == false`, the code returns `-y` which makes it positive, contradicting the intended behavior.

Current problematic logic:

```
let y = if is_largest && choose_largest { y } else { -y };
```

Issue: If `y` is negative (`is_largest == false`) and `choose_largest == false`, the function returns `-y` (positive), but it should return the negative value.

Current Impact: This bug has **no impact** on the current SALT protocol because:

- The `get_point_from_x` function is only called internally within `Element::from_bytes()`
- All calls to `from_bytes()` use `choose_largest = true` (hardcoded)
- The protocol never requests the ”smaller” y-coordinate, so the buggy `else` branch is never executed

Future Library Impact: If this code is used as a standalone library where external callers can specify `choose_largest = false`, this bug would cause:

- Invalid curve point generation when requesting negative y-coordinates
- Cryptographic protocol failures in applications expecting consistent point representation

Fix

```
let y = if is_largest == choose_largest { y } else { -y };
```

This ensures that when `choose_largest == false`, we always get the negative y-coordinate, regardless of whether the initial `y` was positive or negative.

Issue M-2: Missing length validation in `update()` function leading to potential panics [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/62

This issue has been acknowledged by the team but won't be fixed at this time.

Description

The `update()` function in `state.rs` lacks proper length validation for key-value pairs before passing them to `SaltValue::new()`, which can cause panics when the total encoded size exceeds `MAX_SALT_VALUE_BYTES` (94 bytes).

Vulnerable code path:

```
pub fn update<'b>(&mut self, kvs: impl IntoIterator<Item = (&'b Vec<u8>, &'b
↪  Option<Vec<u8>>)>) -> Result<StateUpdates, Store::Error> {
    for (plain_key, plain_val) in kvs {
        match plain_val {
            Some(plain_val) => {
                self.shi_upsert(bucket_id, plain_key.as_slice(),
                ↪  plain_val.as_slice(), &mut state_updates)?;
                // ↓ Calls SaltValue::new() without validation
            }
        }
    }
}
```

The panic occurs in `SaltValue::new()`:

```
pub fn new(key: &[u8], value: &[u8]) -> Self {
    let key_len = key.len();
    let value_len = value.len();
    // ↓ This can panic if key_len + value_len + 2 > 94
    data[2..2 + key_len].copy_from_slice(key);
    data[2 + key_len..2 + key_len + value_len].copy_from_slice(value);
}
```

The upstream validator code may not be checking the total encoded size constraint (`key_len + value_len + 2   94`), allowing oversized data to reach the state management layer.

Fix

Add length validation in the `update()` function:

```rust
pub fn update<'b>(&mut self, kvs: impl IntoIterator<Item = (&'b Vec<u8>, &'b
 ↪  Option<Vec<u8>>)>) -> Result<StateUpdates, Store::Error> {
    let mut state_updates = StateUpdates::default();
    for (plain_key, plain_val) in kvs {
        // Validate total encoded size before processing
        if let Some(plain_val) = plain_val {
            if plain_key.len() + plain_val.len() + 2 > MAX_SALT_VALUE_BYTES {
                return Err(Store::Error::InvalidInput("Key-value pair too large"));
            }
        }

        let bucket_id = hasher::bucket_id(plain_key);
        match plain_val {
            Some(plain_val) => {
                self.shi_upsert(bucket_id, plain_key.as_slice(),
                 ↪  plain_val.as_slice(), &mut state_updates)?;
            }
            None => self.shi_delete(bucket_id, plain_key.as_slice(), &mut
             ↪  state_updates)?,
        }
    }
    Ok(state_updates)
}
```

Issue M-3: Missing domain validation for challenge point in IPA proof [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/65

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `open_point_outside_of_domain` function in `multiproof.rs` fails to validate that the challenge point `t` is outside the evaluation domain [0, 255], breaking the Lagrange coefficient computation when `t` falls within this range.

Vulnerability Detail

When computing multi-point proofs, the protocol generates a random challenge point `t` via `transcript.challenge_scalar(b"t")` that must be outside the polynomial's evaluation domain for the IPA proof to work correctly. However, there's no validation that `t   [0, 255]`.

While the probability is negligible (~2^-245 for a 256-element domain), if `t` falls within [0, 255], the `evaluate_lagrange_coefficients` function will fail when computing `t - i` where `i` equals `t`.

The issue is acknowledged in the code with an unimplemented TODO comment: `// TODO: check that the point is actually not in the domain`.

Impact

The protocol will fail when the challenge point `t` is in the domain, causing the Lagrange coefficient computation to break. This violates the fundamental assumption of the IPA proof that evaluation must occur at a point outside the domain.

Code Snippet

```rust
// multiproof.rs - Line 240-250
pub(crate) fn open_point_outside_of_domain(
    crs: CRS,
    precomp: &PrecomputedWeights,
    transcript: &mut Transcript,
    polynomial: LagrangeBasis,
    commitment: Element,
    z_i: Fr,
) -> IPAProof {
    // TODO: check that the point is actually not in the domain
    let a = polynomial.values().to_vec();
    let b = LagrangeBasis::evaluate_lagrange_coefficients(precomp, crs.n, z_i);  //
    ↪ Breaks if z_i in [0, crs.n)

    crate::ipa::create(transcript, crs, a, commitment, b, z_i)
}

// multiproof.rs - Line 144
let t = transcript.challenge_scalar(b"t");  // No validation that t is outside
↪ domain
let g_3_ipa = open_point_outside_of_domain(crs, precomp, transcript, g_3_x,
↪ g_3_x_comm, t);
```

https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/blob/d5a3cad2f9d4ac796a46927899385f000441d8a7/salt/ipa-multipoint/src/multiproof.rs#L325

Tool Used

Manual Review

Recommendation

Add a validation check in `open_point_outside_of_domain` to ensure the challenge point is not in the domain [0, crs.n). If the point falls within the domain, the function should either panic with a descriptive error message or implement a deterministic remapping strategy to move the point outside the domain.

Issue L-1: Integer overflow in bucket resize capacity calculation [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/60

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`compute_resize_capacity()` contains potential integer overflow vulnerabilities in its arithmetic operations that could lead to incorrect capacity calculations or memory exhaustion during bucket resizing operations.

Vulnerability Detail

`compute_resize_capacity()` performs arithmetic operations on capacity values without overflow protection:

```
fn compute_resize_capacity(current_capacity: u64, target_capacity: u64) ->
  u64 {
    let mut new_capacity = current_capacity;
    while new_capacity < target_capacity {
        new_capacity = new_capacity * 2; // Potential overflow
    }
    new_capacity
}
```

Several overflow scenarios exist:

1. Multiplication Overflow: new_capacity * 2 can overflow when new_capacity exceeds u64::MAX / 2
2. Infinite Loop: If overflow occurs with wrapping arithmetic, the loop may never terminate
3. Invalid Capacity: Overflowed values could result in extremely small capacities due to wraparound

The function is called during bucket expansion operations when the current capacity is insufficient for the number of elements being inserted.

Impact

Memory exhaustion, infinite loop or excessive or too little memory can be allocation to bucket.

Code Snippet

salt/src/state/state.rs:L636-L642

Tool Used

Manual Review

Recommendation

Implement overflow-safe arithmetic.

Issue L-2: Inconsistent power-of-two validation between CRS and IPA [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/66

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The IPA proof creation function panics on non-power-of-two input lengths despite the CRS allowing arbitrary sizes, creating an inconsistency that can cause unexpected runtime failures.

Vulnerability Detail

The `create` function in `ipa.rs` uses `assert!` to enforce that input vector lengths are powers of two, while `CRS::new` accepts any size. This inconsistency means valid CRS instances can trigger panics when used with IPA proofs.

Impact

Applications using arbitrary-sized CRS will experience runtime panics when creating IPA proofs.

Code Snippet

```
// crs.rs - allows any size
pub fn new(n: usize, seed: &'static [u8]) -> CRS {
    let all_points = generate_random_elements(n + 1, seed);
    // No power-of-two restriction
}

// ipa.rs - panics on non-power-of-two
pub fn create(...) -> IPAProof {
    // ...
    assert!(n.is_power_of_two()); // Panic here!
}
```

https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/blob/d5a3cad2f9d4ac796a46927899385f000441d8a7/salt/ipa-multipoint/src/ipa.rs#L93-L99

Tool Used

Manual Review

Recommendation

Either enforce power-of-two restrictions consistently in both CRS creation and IPA proof generation, or handle non-power-of-two cases gracefully by returning proper errors instead of panicking.

Issue L-3: CRS generation panics on duplicate points [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/68

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The CRS generation function panics when duplicate points are detected

Vulnerability Detail

The `CRS::new()` function calls `assert_dedup()` which panics if any duplicate points exist in the generated set. While hash-to-curve collisions are astronomically rare (~1/2^255 probability), the function provides no recovery mechanism and will crash the entire program if duplicates occur.

Impact

Denial of service with no fallback mechanism if duplicate points are generated.

Code Snippet

```
pub fn new(n: usize, seed: &'static [u8]) -> CRS {
    let all_points = generate_random_elements(n + 1, seed);
    let (G, q_slice) = all_points.split_at(n);
    let G = G.to_vec();
    let Q = q_slice[0];

    CRS::assert_dedup(&all_points); // <-- Panics on duplicates

    CRS { n, G, Q }
}

fn assert_dedup(points: &[Element]) {
    let mut map = HashSet::new();
    for point in points {
        let value_is_new = map.insert(point.to_bytes());
        assert!(value_is_new, "crs has duplicated points") // <-- Panic here
    }
}
```

https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/blob/d5a3cad2f9d4ac796a46927899385f000441d8a7/salt/ipa-multipoint/src/crs.rs#L51-L52

Tool Used

Manual Review

Recommendation

Handle duplicates gracefully.

Issue L-4: SALT cannot support EVM variants with non-standard account structures [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/69

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

SALT's `StateAccount` struct is hardcoded for standard Ethereum account layout and cannot support EVM variants like Blast that require different storage structures for features such as dynamic balances.

Vulnerability Detail

The current implementation uses a fixed `StateAccount` struct tailored to standard Ethereum/Mega-EVM. This structure cannot accommodate EVM variants with different account layouts - for instance, Blast requires additional storage space for dynamic balance features and other extensions. Although the development team indicates SALT is designed to be "agnostic to the actual key-value pairs being authenticated," the hardcoded struct approach prevents supporting chains with non-standard account structures without code modifications.

Impact

SALT cannot support EVM chains with extended account structures such as Blast, limiting its compatibility to standard EVM implementations only.

Code Snippet

```
// Current implementation from Blast (example of unsupported EVM)
type StateAccount struct {
    Nonce uint64
    Flags uint8
    Fixed *big.Int
    Shares    *big.Int
    Remainder *big.Int
    Root      common.Hash
    CodeHash  []byte
}

// SALT cannot accommodate different layouts like this
// without modifying the hardcoded struct
```

Tool Used

Manual Review

Recommendation

Consider implementing a configurable encoding approach that can handle variable account layouts across different EVM implementations, aligning with SALT's design principle of being agnostic to key-value pair structures.

Issue L-5: Different byte representations of a point [RESOLVED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/70

Summary

There are two valid uncompressed byte representation of any point. And should be compared carefully. Especially in the case of `Committer::zero`

Vulnerability Detail

Point decoded from compressed empty bytes `[0u8; 32]` will arrive at two torsion point `(0, -1)`. One might accidentaly think that it should be the standard zero `(0, 1)`. So `Committer::zero` representation will not be equal with the point decoded from empty bytes.

Impact

Informational. Salt or IPA parts are checked and they are currently not hitting such case.

Code Snippet

```
assert_eq!(Committer::zero(), Element::zero().to_bytes_uncompressed());

assert_ne!(
    Committer::zero(),
    Element::from_bytes(&[0u8; 32])
        .unwrap()
        .to_bytes_uncompressed()
);
```

Tool Used

Manual Review

Recommendation

Always compare points or objects that are indeed points in `Element` form never compare in byte representations

Issue L-6: Paralel addition of deltas [RESOLVED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/71

Summary

Parallelized addition of deltas is much faster on author's machine. In some cases in salt side `Committer::add_deltas`are called in single thread routine.

Vulnerability Detail

Not a vulnerability

Impact

5x performance gain

Code Snippet

```
delta_indices
            .par_iter()
```

Tool Used

Manual Review

Recommendation

Function potentially named par_add_deltas could be introduced to enjoy faster delta addition especially when called from single threaded routine

Issue L-7: Use arkworks mixed addition for non x86 targets [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-08-megaeth-salt-aug-26th/issues/72

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Arkwoks has support for mixed addition operation. For non x86 targets custom addition implementation could be replaced.

Vulnerability Detail

Not a vulnerability

Impact

Less number of lines to maintain

Code Snippet

```
#[cfg(not(target_arch = "x86_64"))]
fn add_affine_point(result: &mut EdwardsProjective, p2_x: &Fq, p2_y: &Fq) {
       *result += EdwardsAffine::new_unchecked(*p2_x, *p2_y);
}
```

Tool Used

Manual Review

Recommendation

Moving to arkworks API for this operation could be considered

Disclaimers