Commit

✓ Fix potential integer overflow in hash container create/resize

Browse files

```
The sized constructors, reserve(), and rehash() methods of
absl::{flat,node}_hash_{set,map} did not impose an upper bound on
their size argument. As a result, it was possible for a caller to pass
a very large size that would cause an integer overflow when computing
the size of the container's backing store. Subsequent accesses to the
container might then access out-of-bounds memory.

The fix is in two parts:

1) Update max_size() to return the maximum number of items that can be
stored in the container

2) Validate the size arguments to the constructors, reserve(), and
rehash() methods, and abort the program when the argument is invalid

We've looked at uses of these containers in Google codebases like
Chrome, and determined this vulnerability is likely to be difficult to
exploit. This is primarily because container sizes are rarely
attacker-controlled.

The bug was discovered by Dmitry Vyukov <dvyukov@google.com>.

PiperOrigin-RevId: 718841870
Change-Id: Ic09dc9de140a35dbb45ab9d90f58383cf2de8286
```

master
20250127.0 ··· 20250127.rc1

derekmauro authored and **copybara-github** committed last month 1 parent 5f8d605 commit 5a0e2cb

Showing **3 changed files** with **38 additions** and **1 deletion**.

Whitespa... Ignore whitespace Split Unifi...

Filter changed files

absl/container/internal

5 absl/container/internal/raw_hash_set.cc

```
24  24    #include "absl/base/config.h"
25  25    #include "absl/base/dynamic_annotations.h"
26  26    #include "absl/base/internal/endian.h"
    27  + #include "absl/base/internal/raw_logging.h"
27  28    #include "absl/base/optimization.h"
28  29    #include
          "absl/container/internal/container_memory.h"
```

```
29   30   #include
          "absl/container/internal/hashtablez_sampler.h"
661  662    return target.offset;
662  663  }
663  664
     665  + void HashTableSizeOverflow() {
     666  +   ABSL_RAW_LOG(FATAL, "Hash table size overflow");
     667  + }
     668  +
664  669  }  // namespace container_internal
665  670  ABSL_NAMESPACE_END
666  671  }  // namespace absl
```

26 absl/container/internal/raw_hash_set.h

```
1226 1226    // Given the capacity of a table, computes the
          total size of the backing
1227 1227    // array.
1228 1228    size_t alloc_size(size_t slot_size) const {
     1229  +    ABSL_SWISSTABLE_ASSERT(
     1230  +        slot_size <=
     1231  +        ((std::numeric_limits<size_t>::max)() -
          slot_offset_) / capacity_);
1229 1232      return slot_offset_ + capacity_ * slot_size;
1230 1233    }
1231 1234
1544 1547    return n ? ~size_t{} >> countl_zero(n) : 1;
1545 1548  }
1546 1549
     1550  + template <size_t kSlotSize>
     1551  + size_t MaxValidCapacity() {
     1552  +   return
          NormalizeCapacity((std::numeric_limits<size_t>::ma
          x)() / 4 /
     1553  +                           kSlotSize);
     1554  + }
     1555  +
     1556  + // Use a non-inlined function to avoid code bloat.
     1557  + [[noreturn]] void HashTableSizeOverflow();
     1558  +
1547 1559  // General notes on capacity/growth methods below:
1548 1560  // - We use 7/8th as maximum load factor. For 16-
          wide groups, that gives an
1549 1561  //   average of two empty slots per group.
2645 2657        :
          settings_(CommonFields::CreateDefault<SooEnabled()
          >(), hash, eq,
2646 2658                    alloc) {
2647 2659      if (bucket_count > DefaultCapacity()) {
     2660  +      if (ABSL_PREDICT_FALSE(bucket_count >
     2661  +
          MaxValidCapacity<sizeof(slot_type)>())) {
     2662  +        HashTableSizeOverflow();
```

```diff
+      }
       resize(NormalizeCapacity(bucket_count));
     }
   }
     ABSL_ASSUME(cap >= kDefaultCapacity);
     return cap;
   }
-  size_t max_size() const { return (std::numeric_limits<size_t>::max)(); }
+  size_t max_size() const {
+    return CapacityToGrowth(MaxValidCapacity<sizeof(slot_type)>());
+  }

   ABSL_ATTRIBUTE_REINITIALIZES void clear() {
     if (SwisstableGenerationsEnabled() &&
     auto m = NormalizeCapacity(n | GrowthToLowerboundCapacity(size()));
     // n == 0 unconditionally rehashes as per the standard.
     if (n == 0 || m > cap) {
+      if (ABSL_PREDICT_FALSE(m > MaxValidCapacity<sizeof(slot_type)>())) {
+        HashTableSizeOverflow();
+      }
       resize(m);

       // This is after resize, to ensure that we have completed the allocation
     const size_t max_size_before_growth =
         is_soo() ? SooCapacity() : size() + growth_left();
     if (n > max_size_before_growth) {
+      if (ABSL_PREDICT_FALSE(n > max_size())) {
+        HashTableSizeOverflow();
+      }
       size_t m = GrowthToLowerboundCapacity(n);
       resize(NormalizeCapacity(m));

```

8 absl/container/internal/raw_hash_set_test.cc

```diff
  }
}

+TEST(Table, MaxSizeOverflow) {
+  size_t overflow = (std::numeric_limits<size_t>::max)();
+  EXPECT_DEATH_IF_SUPPORTED(IntTable t(overflow), "Hash table size overflow");
+  IntTable t;
```

```
                3744 +   EXPECT_DEATH_IF_SUPPORTED(t.reserve(overflow),
                       "Hash table size overflow");
                3745 +   EXPECT_DEATH_IF_SUPPORTED(t.rehash(overflow),
                       "Hash table size overflow");
                3746 + }
                3747 +
3740            3748   }  // namespace
3741            3749   }  // namespace container_internal
3742            3750   ABSL_NAMESPACE_END
```

0 comments on commit `5a0e2cb`

Please sign in to comment.