BackupRefPtr

author: glazunov@
last updated: 2020 December 16
status: **PUBLIC**
self link: go/backuprefptr

BackupRefPtr is a proposal for an implementation of MiraclePtr that provides deterministic protection against use-after-free issues.

Background

What is MiraclePtr? => See go/miracleptr.

What are other proposals? => See go/designdoc_PointerSafety.

Motivation

Currently, the most promising MiraclePtr proposals rely on pointer tagging for probabilistic protection. One of the biggest potential problems with this approach is speculative side-channel attacks. For example, even with 16-bit tags, sroettger@'s RIDL exploit for Chrome can be re-used to find a "tag collision" and defeat the protection in reasonable time using a variant of a birthday attack. Furthermore, significantly weaker "1-bit" side-channel attacks, which are currently impractical to use against Chrome, can defeat pointer tagging if they can be turned into an oracle that reports whether a given pointer has (speculatively) passed the tag check.

Therefore, we'd like chrome-memory-safety@ to consider using one of the deterministic MiraclePtr proposals, which are immune to side-channel attacks. This document describes a reference-counting-based implementation and mostly borrows ideas from CheckedPtr2, SafePtr, and BorrowPtr.

Design

BackupRefPtr is a thread-safe reference counting pointer class, which is intended to serve as a replacement for the regular raw pointer type. The reference count is stored as per-allocation PartitionAlloc metadata (the same way as the tag in CheckedPtr2). If after the destruction of a protected object the corresponding reference count doesn't equal zero, the allocation slot is poisoned (filled with a bit-pattern that will likely lead to a crash if the destroyed "object" is accessed) and reclaiming the memory is delayed. This situation can happen when:

- There's an actual UAF bug.

- A dangling BackupRefPtr exists, but it will never be dereferenced by its owning object. The pointee memory will be freed once all the pointers get destroyed.
- An object containing a BackupRefPtr has been leaked.

The slot is returned to the free list if the reference count reaches zero afterwards.

A rough pseudo-code outline. The actual implementation is in base/checked_ptr.h.

```
template <typename T>
class BackupRefPtr {
  BackupRefPtr(T* ptr) : ptr_(ptr) {
    if (!isSupportedAllocation(ptr))
      return;

    atomic_int& ref_count = *(cast<atomic_int*>(ptr) - 1);
    CHECK(++ref_count);
  }

  ~BackupRefPtr() {
    if (!isSupportedAllocation(ptr_))
      return;

    atomic_int& ref_count = *(cast<atomic_int*>(ptr) - 1);
    if (--ref_count == 0) // needed in case the BackupRefPtr outlives
                          // its pointee and has to free the slot
      PartitionAlloc::ActuallyFree(ptr_);
  }

  T* operator->() { return ptr_; }

  T* ptr_;
};

void* Alloc(size_t size) {
  void* ptr = ActuallyAlloc(size);
  if (isSupportedAllocation(ptr)) {
    int& ref_count = *(cast<int*>(ptr) - 1);
    ref_count = 1; // We need to set the reference count to one initially
                   // otherwise |~BackupRefPtr| can trigger deallocation of
                   // an object that's still alive.
  }
  return ptr;
}

void Free(void* ptr) {
  if (isSupportedAllocation(ptr)) {
    atomic_int& ref_count = *(cast<atomic_int*>(ptr) - 1);
    if (ref_count != 1)
      memset(ptr, 0xcc, getAllocationSize(ptr));
```

```
      // It's important to call |memset| before modifying the reference count
      // to prevent possible races between |memset| and |ActuallyFree| in
      // another thread.

    if (--ref_count != 0)
      return;
  }

  ActuallyFree(ptr);
}
```

Originally, we considered using a two-byte saturating integer for the count to reduce memory overhead, however, the alignment requirement placed on the allocator let us have a simpler implementation relying on a regular 4-byte count with an overflow check.

The benefits of the approach are:

- BackupRefPtr provides deterministic protection, which can't be bypassed with side-channel attacks.
- It's transparent i.e. doesn't require classes to inherit from (hypothetical) BackupRefCountedBase (same as CheckedPtr2).
- The dereference operator incurs no overhead.
- As lukasza@ has pointed out, lack of safety checks on dereference (or "extraction" / cast to a raw ptr) means that BackupRefPtr reduces some incompatibilities between MiraclePtr and raw pointers.
- Dangling dereferences may be protected even when the dereference doesn't go through BackupRefPtr (e.g. unlike CheckedPtr2, BackupRefPtr may protect a dangling |this| and/or a dangling const-ref). (Protection depends on the existence of a BackupRefPtr that points to the freed memory.)

The downsides are:

- The performance overhead of constructing/destructing a BackupRefPtr is worse than for CheckedPtr2. As we don't intend to use MiraclePtr for on-stack variables, hopefully, all BackupRefPtrs should be "long-lived" enough, and, given the performance gain from dereferences, the total negative effect shouldn't be too terrible compared to CheckedPtr2.

- Several code patterns, which are safe to use with raw pointers, can cause (indefinitely) delayed deallocations. For example:

```
class A {
 A(T* ptr) : ptr_(ptr), is_valid_(true) { }

  ContextDestroyed() { is_valid_ = false; }
```

```
  DoSomething() {
    if (is_valid_)
      ptr->DoSomethingElse();
  }

  MiraclePtr<T> ptr_;
  bool is_valid_;
};
```

Another case is where a pointer is stored but never dereferenced:

```
class B {
 B(T* ptr) : ptr_(ptr) { }

  RunObserverCallback(T* ptr) {
    if (ptr == ptr_)
      DoSomethingElse();
  }

  MiraclePtr<T> ptr_;
};
```

However, we should be able to statically detect such pointers (for example, with CodeQL) and replace them with MiraclePtrNoDereference<T> in a semi-automated fashion.

- Similarly to std::shared_ptr, the BackupRefPtr class itself isn't thread safe, i.e. if multiple threads modify the same BackupRefPtr object without synchronization, a data race will occur and the reference count will become inconsistent. We assume that each such case is a bug on its own since modifying a raw pointer in a similar manner (i.e. the situation before the MiraclePtr rewrite) would also constitute a race condition, although it wouldn't necessarily lead to memory corruption.

- Pointer laundering. Consider the following scenario:

    1. `A` is the last BackupRefPtr to an allocation, the allocation is already quarantined.
    2. `B` is a raw pointer variable (for example, on the stack), `A` is assigned to `B`.
    3. `A` is released, the allocation is removed from the quarantine.
    4. Dangling `B` is used.

  Tag-based MiraclePtr implementations are immune to the attack because they check the tag on every dereference and would crash in step two. To address the issue in BackupRefPtr, we'd need to check whether the allocation is "alive" whenever the smart pointer is converted into a raw ptr. It's unclear if the performance overhead of such a change would be acceptable.

There's also a version of the attack that affects CheckedPtr2/MTECheckedPtr, in which the allocation is put under quarantine only after `B` is assigned. Most likely, the only way to avoid the issue completely is to make every single pointer "managed" either through annotation or scanning.

Implementation variants

It's conceivable that an attacker can abuse even a poisoned allocation in rare cases. For example:

```
class A {
  B* DoSomething() {
    return base_[index_]; // poison_value + posion_value * sizeof(B)
                          // by luck points to valid memory
  }

  B* base_;
  size_t index_;
};

MiraclePtr<A> ptr;

// ptr is dangling at this point

B* b = ptr->DoSomething();
```

We can modify BackupRefPtr to crash on dereference if we have a dedicated bit in the metadata to store the quarantine state of an allocation. However, in practice it's very unlikely that a properly poisoned object can be exploited, and increasing overhead of the dereference operation is undesired. Perhaps, it makes sense to randomly choose the poison value from a small predefined set.

BackupRefPtr can support pointers to the middle of an allocation if they share the same per-slot reference count. This comes with an additional cost for arithmetic operations needed to determine the beginning of the allocation. The CheckedPtr2 constructor also has to perform the same computation. We can get rid of it in the BackupRefPtr destructor if we store the offset in the top bits of the pointer (same as in CheckedPtr3) and mask them off on dereference.

Biased reference counting

It's likely possible to avoid most atomic reference counting with another space-time tradeoff. If the assumption that the majority of objects are only accessed from a single thread (which is the same as their creator thread) is correct, we can use a pair of counters: a non-atomic one exclusively for the creator thread, and an atomic one for the rest:

```
struct RefCount {
  int32_t creator_tid;
  int32_t creator_count;
  atomic_int32_t shared_count;
};

BackupRefPtr(T* ptr) : ptr_(ptr) {
  RefCount& ref_count = *(cast<RefCount*>(ptr) - 1);
  if (GetCurrentThreadID() == ref_count.creator_tid) {
    ++ref_count.creator_count;
  } else {
    ++ref_count.shared_count;
  }
}
```

The corresponding destructor is more complex, but it also doesn't touch the atomic counter in the fast path. This solution depends on having an efficient GetCurrentThreadID implementation for every major platform. The details of the same algorithm implemented for Swift are available in this paper.