SafePtr
(concurrent use-after-free sanitization)

author: haraken@

2020 Mar 9

Status: PUBLIC

Summary: *This document proposes a high-performance mechanism to change use-after-free on the browser process to crashes. This is an alternative proposal of UnownedPtr.*

[Note: hpayer@ has a similar proposal.]

Problem: Use-after-free in the browser process

See adetaylor@'s analysis of Chrome security bugs in 2014 - 2019 and the MiraclePtr Security Impact.

TL;DR is: our primary goal is to **massively reduce use-after-free bugs in the browser process** but we can apply the same technique to the renderer process as well. In other words, our goal is to **massively reduce use-after-free bugs in Chromium**.

Possible solutions

We discussed a couple of possible solutions to the problem:

(a) Migrate the majority of objects on the browser process to Oilpan (discussion)
(b) Rewrite (parts of) the browser process code in a safer language such as Rust and Swift (proposal)
(c) Replace raw pointers with UnownedPtr (which crashes when the pointer is dereferenced after the target object is gone)

(a) is multiple years of engineering work. (b) is even more. (c) is more realistic but there is a concern about the performance cost and the coverage.

This document proposes ***SafePtr*** as an alternative option to (c). In terms of what it does, SafePtr is the same as UnownedPtr. It just crashes when the pointer is dereferenced after the target object is gone. **SafePtr can be viewed as an alternative implementation of UnownedPtr, which has a different performance / memory / code complexity characteristics**. I am not fully confident which is better.

[Note: SafePtr / UnownedPtr is complementary to (a) or (b). It just changes use-after-free to crashes (which will be sufficient in common cases). If we want to fix potential use-after-free, we still need (a) or (b). SafePtrs / UnownedPtrs are useful to sanitize use-after-free only when the code is written in a way in which use-after-free is rare. Otherwise, we need to consider introducing Oilpan or Rust. For example, it is nearly impossible to implement Blink without having a memory management system that can deal with circular references. PDFium is in this category as well.]

Proposal

Developer API

Using SafePtr is very easy. Developers just need to replace a raw pointer with SafePtr.

Before:

```
class SomeObject {
  T* ptr_;
};
```

After:

```
class SomeObject {
  SafePtr<T> ptr_;
};
```

This is a mechanical change and can be done by a clang plugin (if it is okay to replace everywhere from the performance / memory perspective). I'm assuming that we replace only not-on-stack raw pointers with SafePtrs. On-stack raw pointers are less likely to cause use-after-free and thus stay as is.

Concurrent sanitization

Overview

The goal is to crash the process when a SafePtr is dereferenced after the target object is destructed. This can be implemented as follows.

1. Replace malloc with multi-threaded PartitionAlloc. Make sure that all objects pointed to by SafePtrs are allocated by PartitionAlloc.
2. Keep track of the list of all SafePtrs.
3. When an object is destructed, poison the memory region and move it to a ***pending list*** of PartitionAlloc (not a free list). If code dereferences a SafePtr pointing to the pending list, it crashes.
4. When the size of the pending list reaches some threshold, kick off a ***concurrent sanitizer***. The concurrent sanitizer iterates all SafePtrs. If there is any SafePtr pointing to the pending list, crash the process. Move the pending list to the free list.

Step 1: Replace malloc with multi-threaded PartitionAlloc

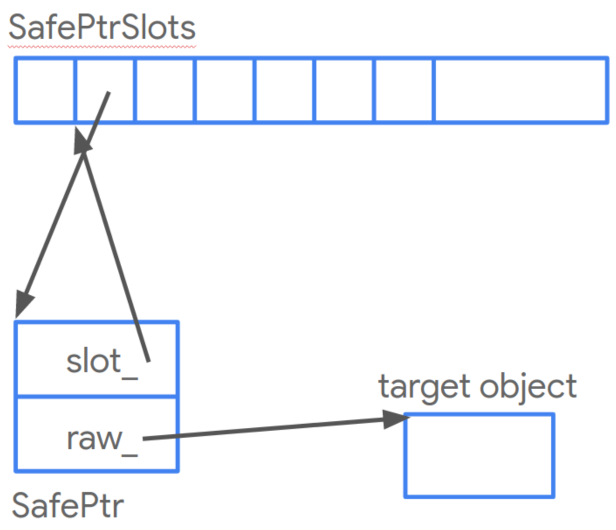
This is on-going work because it has a lot of other benefits.

Step 2: Keep track of all SafePtrs

SafePtr can be implemented as follows. In essence, SafePtr is similar to Oilpan's Persistent handle (See how Oilpan's PersistentNode and PersistentRegion are implemented).

```
template <T>
class SafePtr {
  T* raw_;  // The pointer to the target object.
  SafePtrSlot* slot_;  // The slot on which the SafePtr is stored.
}
```

SafePtrs are stored in SafePtrSlots (a fixed array of slots). SafePtrRegion manages the list of SafePtrSlots and the free list of the slots.



SafePtr's constructor needs the following overhead:

```
SafePtr(T* raw) {
  raw_ = raw;
  // In common cases, AllocateSlot() just returns the free list head.
  slot_ = TLS::Current()->GetSafePtrRegion()->AllocateSlot(this);
}
```

SafePtr's destructor needs the following overhead:

```
~SafePtr() {
  // FreeSlot() just returns the slot to the free list.
  TLS::Current()->GetSafePtrRegion()->FreeSlot(slot_);
}
```

On the other hand, **SafePtr's dereference is just a pointer dereference**.

I am worried about the overhead of TLS::Current() because thread local storage is slow on some platforms. This issue needs to be fixed anyway because TLS::Current() is used by a ton of other places, including the multi-threaded PartitionAlloc. When we replace malloc with the multi-threaded PartitionAlloc, every object allocation and deallocation needs to look up TLS::Current() to access a threaded cache.

Step 3: Object destruction

When an object is destructed, poison the object's memory region with 0xBADC0DE and move it to a pending list of PartitionAlloc (not a free list). This can be easily implemented in PartitionAlloc::Free(). The only difference between the existing PartitionAlloc::Free() and the new PartitionAlloc::Free() is that the latter uses 0xBADC0DE instead of 0x0.

Step 4: Concurrent sanitization

When the size of the pending list reaches some threshold, we kick a sanitizer using a concurrent thread. The concurrent sanitizer iterates all SafePtrs. If there is any SafePtr pointing to the pending list, we crash the process.

At this point, all memory regions in the pending list are filled with 0xBADC0DE. The sanitization works like this:

```
SafePtr::Sanitize() {
  if (UNLIKELY(raw_ && *raw_ == 0xBADC0DE)) {
    // raw_ is likely to be pointing to the pending list.
    if (InPendingList(raw_)) {
      CRASH();
    }
  }
}
```

[Note: There may be a valid use case where *raw_ is pointing to a memory region that happens to have a value of 0xBADC0DE. So we need to check whether raw_ is really included in the pending list. It's fine that InPendingList() is slow (e.g., simply scan the pending list) because it is rare to hit the branch.]

Finally, the concurrent sanitizer unpoisons the pending list and moves the pending list to the free list.

Note that PartitionAlloc::Free() needs to use atomicops::Release_Store() to move a memory region to the pending list (because the access to the pending list races with the concurrent sanitizer). PartitionAlloc::Allocate() needs to use atomicops::Acquire_Load() to allocate a memory region from the free list (because the access to the free list races with the concurrent sanitizer). The overhead of atomicops::Release_Store()/Acquire_Load() will be negligible.

Performance / memory overhead

The performance / memory overhead of SafePtrs is summarized as follows:

(1) The memory overhead of SafePtr. It adds one word per SafePtr.
(2) The performance overhead of SafePtr's construction and destruction.
(3) The memory overhead of the pending list and the SafePtrRegion.
(4) The performance overhead of the concurrent sanitizer. It needs to scan all SafePtrs and then unpoisons all entries in the pending list.
(5) The performance overhead of atomicops::Release_Store/Acquire_Load() in PartitionAlloc::Free/Allocate().

I'm not worried about (3), (4) or (5). (3) can be reduced by optimizing the threshold to kick the concurrent sanitizer. (4) runs concurrently and does not affect the main thread time. (5) will be negligible.

I'm not really worried about (1) because the overhead is the same as WeakPtr, UnownedPtr etc. If the memory overhead becomes an issue on some specific object, we can just stop using SafePtr on the object.

I'm worried about (2). One mitigation would be to not use SafePtrs on performance-sensitive objects.

Ideally it's awesome if we can replace all raw pointers with SafePtrs using a clang plugin but it won't be realistic. At least, **we should aim at replacing all base::Unretained() with SafePtrs**.

Discussion: SafePtr vs. UnownedPtr

UnownedPtr will be implemented as WeakPtr. The only difference is that UnownedPtr crashes when the target object is already destructed.

WeakPtr:

```
T* get() {
  return ref_.IsValid() ? ptr_ : nullptr;
}
```

UnownedPtr:

```
T* get() {
  if (UNLIKELY(!ref_.IsValid()) CRASH();
  return ptr_;
}
```

The next table compares pros and cons of SafePtr and UnownedPtr.

	SafePtr	UnownedPtr
Developer-facing API	Very easy. Just replace T* with SafePtr<T>.	Need to add a WeakPtrFactory. Need to call weak_ptr_factory_.GetWeakPtr() to create a WeakPtr.
Allocation overhead	A TLS lookup + a slot allocation from the free list head.	Cheap. Just increment a ref count of the shared state.
Free overhead	A TLS lookup + a slot deallocation to the free list	Cheap. Just decrement a ref count of the shared state.
Pointer dereference	Cheap. Same as a single pointer dereference.	One dereference of the shared state + one if branch + one dereference of the pointer.
Memory overhead per pointer	One additional word	One additional word
Additional memory overhead	Pending list + SafePtrSlotRegion	Shared states
Others	This will delay reusing the destructed objects. It might regress cache locality but will improve security.	Easy to experiment. Mostly the same as WeakPtr.

IMHO it is hard to say which is better at this point.