What’s Up With Pointers

With Special Guest Dana

Intro

We’re lucky to have Dana as our first guest on this series! She’s a base and C++ owner and is currently working on introducing Rust to Chromium. Previously, she was part of bringing C++11 support to the Android NDK, and then to Chrome.

What Can Go Wrong

- Pointers are a common cause of security problems in Chrome
- The most common type is a use-after-free (UaF). You have a pointer to an object (i.e. a goat), that object gets deleted, then something else (a cow) gets allocated in the same place. The pointer still thinks it’s pointing to a goat, and that misunderstanding can be used by attackers
- Other examples of memory safety issues that can happen with pointers are array out of bounds errors, but we’ll primarily focus on use-after-frees.
- Check out Dana’s talk Life of a Vulnerability, which includes a use-after-free in the attack

Should you ever use a native pointer?

- In C++, a **native pointer** looks like Foo*
- You can avoid use-after-frees by never using native pointers, but there are performance trade-offs, so this isn’t always feasible
- The style guide says raw pointers (and references) are ok for giving access to an object, e.g. passing as a function parameter
- You **should not use native pointers** in the following cases:
    - As fields in objects, since these are the ones that tend to lead to use-after-frees
    - To express ownership - when ownership gets passed around, this is very difficult to do correctly. If you have to do this, use a std::unique_ptr

What are smart pointers

- AKA a pointer like object
- A smart pointer is a class that holds a pointer inside it and mediates access to that pointer in some way
- E.g. a std::unique_ptr says I own the pointer and will delete the pointer when I go away, but you can use it, by using the -> operator and * operator
- Adds something to a native pointer, while still giving you access to it in some way

- There are a lot of these types in Chromium, they'll be covered below

Types of smart pointers and pointer like objects

std::unique_ptr<Foo>

- This expresses **unique** ownership to an object in the heap
- Holds your pointer and deletes it when it goes out of scope
- Can't be copied, because it's unique ownership, but it can be moved
- Is the size of a pointer, the object it points to lives elsewhere in the heap

absl::optional<Foo>

- **Use when** you want to return a value sometimes, either there's no value to give back or in case of an error
  - This is more clear than using a native pointer and bool to express the same thing
- This type is a pointer-like object because it provides the -> and * operators, even though it holds an object inside of it by value, rather than a pointer
- That means the object is held in the space allocated for the optional, aka the size of the optional is the size of the thing it's holding, plus a flag.
- Similar to a std::unique_ptr because it uniquely holds that object
- It's unlike std::unique_ptr in a few ways
  - It is copyable if the object inside is copyable
  - Doesn't need a heap allocation, the object in the optional and the optional itself is all on the stack. This can be a pro or a con, depending on what you're optimizing for
  - Have to include the type in the optional, a forward declaration isn't enough since the optional needs to know the size of the object
- Used to be base::Optional, which was a backport. Now we use absl::optional. This is Chrome's alternative to std::optional. We eventually want to move to std::optional, when we can enable the same safety checks in it that we have in absl.

base::scoped_refptr<Foo>

- **Use carefully! Refcounting is hard!**
- It's an owning smart pointer, so owns a pointer to something allocated in the heap
- Gives shared ownership of the underlying object, since it can be copied.
- When all scoped_refptrs pointing to the same object are gone, that object gets destroyed
- Chrome's equivalent of std::shared_ptr, with some differences
  - scoped_refptr requires type held inside to inherit from RefCounted or RefCountedThreadSafe, because the ref counting happens in the object, where in shared_ptr it happens outside the object

- The ThreadSafe part of RefCountedThreadSafe is important because if there are scoped_refptrs to the same object on different threads, they could race and be wrong which can lead to a double free. With RefCountedThreadSafe, you get atomic refcounting, which makes it thread safe
    - Be sure you know what thread has the last reference to the object, because otherwise it can behave in flaky ways

base::WeakPtr<Foo> and base::WeakPtrFactory<Foo>

- Main purpose is for asynchronous work, which is most work in Chrome
- The issue with async work is that there isn't a continuous stack frame. You can do work on a stack frame, you go away for a while and when you come back, the state of that stack frame is gone.
    - Any state you want to keep across tasks needs to be bound with the task, but that's risky with pointers, as the use-after-free discussion earlier highlights
- WeakPtrFactory provides a side channel that watches the object and tells WeakPtr if the underlying object is still there.

base::SafeRef<Foo>

- **Use when** you want to guarantee that Foo exists and is valid
- If the assumption that the object is valid is broken, then the process is guaranteed to terminate safely and make a crash report. That's not ideal, but it's better than a security bug and ensures we hear about the bug.
- Makes human understanding easier because it reduces possible code branches. Instead of Foo being either null or non-null, it's always non-null and reduces the number of states your code can be in. Always prefer fewer states.
- Built on top of WeakPtr, so it does require a WeakPtrFactory to be present.

raw_ptr<Foo>

- **Use when** you have a pointer as a class member (unless one of the other types has features you're looking for)
- Keeps a reference count in the memory allocator (Chrome' has its own called PartitionAlloc)
- If the object is deleted, the allocator will 'poison' the memory that object occupied and keep the memory around so it's not reused, while there is a raw_ptr pointing to it. This reduces the risk/impact of a use-after-free bug.
- These currently aren't turned on outside the browser process, but you should use them any time you have class members in the browser process, including in code that is used in multiple processes.
- These have weak refcounting, unlike scoped_refptr which is strong refcounting
    - A strong refcount means the refcount owns the object. When the count goes to 0, the object gets deleted

    - This weak refcount keeps the memory allocated, but doesn't keep the object in the memory alive, so they don't affect the behaviour of the code
    - These can exist at the same time (e.g. a raw_ptr to an object owned by scoped_refptr).
- This is part of the MiraclePtr project. Here's a talk about it

base::expected<Foo, Error>

- Use for return values; allows for adding a detailed error result
    - E.g. for file IO, which might fail for a variety of reasons, like the file not existing or not having permissions
- Replacement for exceptions (which we don't enable in Chrome)

Rust?

- A newer language with cool properties, which give the language memory safety to eliminate whole classes of errors
    - Use-after-frees don't exist!
- This kind of language makes sense for Chrome, which runs arbitrary (and thus possibly malicious) code

Join In

- If you want to get involved, check out cxx@chromium.org and rust-dev@chromium.org mailing lists,, along with #base #cxx, and #rust on Slack