Mojo/IPC Security Review Reference

dcheng, with ajgo, mattdr, ortuno, rsesek, tsepez, wfh, and many more.
Updated: 2022-May-27

Background

This document is intended as an outline that you can use when reviewing IPC changes. The first two sections cover concepts that are useful to keep in mind throughout the entire review; the later sections focus more on specific details to look for on a per-file basis.

As an IPC reviewer, you have two goals:

- **Prevention: looking for (security) bugs in IPC changes.** By definition, IPC crosses process boundaries, and Chrome uses process isolation as a security boundary. Bugs in IPC can allow an attacker to pivot from a less trustworthy process to a more trustworthy process that is privileged, increasing the bug's severity.
- **Mitigation: making sure IPC changes follow best practices.** IPCs should be hard to implement or use incorrectly. This protects against bugs that slip through IPC review and are released to users, and it makes it less likely that future changes to IPC implementations or interfaces will introduce bugs.

Basics

At minimum, CLs should include a bug number and a good commit description. Describe the what and the why of the change, not how the change is implemented. Link any relevant background reading from the bug and/or the CL description.

CLs should follow the Security section of the Mojo style guide. This document goes into more detail about those requirements.

Review Flow

1. Review the Mojo interface changes. Understand what is changing and why it is changing. Issues in this area are often of the type "this IPC is overly general/broad/powerful".
2. Review implementation changes. Look for ways that implicit assumptions and invariants can be violated (e.g. what if the IPC sender is malicious?). Check if those implicit assumptions and validations are explicitly handled. Issues in this area range from arithmetic overflows to use-after-frees to confusing an internal state machine.
3. Look for ways to hoist as many explicitly required checks as possible out of the implementation code. Encourage code to rely on centralized validation by constraining

behavior using separate interfaces and/or strict typing. Issues in this area may not be immediate bugs, but may increase the risk of future bugs.

Ask Questions

An IPC review often requires a collaborative effort with the CL author. You may be assigned as a reviewer for CLs where you are not a domain expert. This is okay: if something is unclear to you, it will often be unclear to other people trying to understand the code as well.

Instead of exhaustively studying the CL to try to figure out the details, ask the CL author to help clarify any points of confusion. The answers to those questions may be valuable context to add in code or mojom comments. For example, if the code makes an assumption about two arrays always being the same size, ask the CL author to highlight what is responsible for ensuring that assumption always holds–and add references in comments to make it easier to trace the safety for future readers.

The goal is to ensure that Mojo changes are sufficiently well-described with comments that an unfamiliar reader can:

- Understand the high-level purpose of the Mojo changes, and how the different cross-process components work together.
- Know who to contact or where to look for more details.

Ensure the IPC makes sense

IPCs are often used to provide a capability to a sandboxed process that it would otherwise not have. For example, the renderer process's sandbox prevents it from directly using various socket APIs, so the work of loading network resources is driven over IPC instead.

As a reviewer, understand what holes a new or changed IPC punches through security boundaries such as the sandbox, and ask yourself if the IPC makes sense, or if the IPC is overly broad. Allowing the renderer process to create a file with an appropriately-constrained path? Possibly OK. Allowing the renderer process to create a file at an arbitrary path? Definitely not OK.

If you cannot easily make this determination, please reach out to the CL author to see if their feature has already been security reviewed (note: this is distinct from the IPC review process discussed in this reference), or if there is additional context/discussion/documentation. If there is sufficient security ambiguity, it is OK and strongly encouraged to wait until that ambiguity is resolved before LGTMing.

Determine which side of the IPC is trustworthy

In almost all IPCs, there is a trust and privilege difference between the two processes that are communicating. As an IPC reviewer, it is crucial that you can determine which side is considered trustworthy.

Assume the untrustworthy side is compromised

For the rest of the review, you should assume that the attacker has full control over the untrustworthy process. Even if something cannot “normally” happen in the code as written, an attacker with full control over a process can make arbitrary IPCs. When evaluating an interface, its implementation, or its usage:

- Is the interface supposed to have a restricted scope, e.g. is it implementing an experimental web API that is still behind a runtime flag? Does the browser process side validate that the runtime flag is actually enabled? Example bug: https://crbug.com/1270358
- What would happen if unexpected values are passed in the input or output arguments, e.g. what happens if a size is a negative number?
- What would happen if method calls are out of order, repeated, or improperly matched up?
- Are there IDs (e.g. an integer index into a map) or values (e.g. origins, file paths) used to identify a resource? Can the IDs be crafted or spoofed? Example bug: https://crbug.com/352395
- Are there indexes, offsets, or sizes being passed over IPC? What guarantees that these values will be in range?
- Does the more trustworthy side otherwise violate the rule of 2 in its handling of untrustworthy input, e.g. by parsing untrustworthy JSON in the browser process using the C++ base::JSONReader class?

If any of these assumptions can be violated by an untrustworthy caller, can that lead to a security bug?

Follow (or ask about) the flow of data passed over IPC, e.g. if a method takes a string `name` parameter, does that end up calling `fopen(name)`? What happens if an untrustworthy caller takes the initiative to specify a more interesting name, like `/etc/passwd`?

Design Considerations

IPCs should have well-defined constraints and a clear purpose

By definition, IPCs are an interface between two processes; it is important for that boundary to be easy to understand.

This was already previously mentioned but is important enough to bear repeating: **make sure that an added or changed IPC makes sense**. IPCs are often introduced to get around sandbox restrictions, but an IPC that allows a renderer process to obtain a handle to any arbitrary file in the user's filesystem would completely defeat the point of having a renderer sandbox. Such an IPC is never allowed in Chrome.

Work with the CL author to understand what capabilities need to be exposed over IPC and ensure that those capabilities are appropriately constrained.

Create logical scopes with interfaces

A non-exhaustive listing of common patterns is included below. Following these patterns makes code more robust against bugs and intentionally malicious misuse.

Minimize "statefulness" within an interface

Avoid:
```
interface VideoDecoderService {
  // Must be called exactly once before any other method is used.
  Initialize(VideoDecoderParams params);
  DecodeFrame(array<uint8> data) => (VideoFrame frame);
};
```

Prefer to vend an interface to express this state transition instead:
```
interface VideoDecoderService {
  CreateDecoder(
      VideoDecoderParams params,
      pending_receiver<VideoDecoder>);
};
interface VideoDecoder{
  DecodeFrame(array<uint8> data) => (VideoFrame frame);
};
```

With the preferred pattern, a malicious caller cannot intentionally violate the ordering constraints. Example bug: https://crbug.com/999311

Avoid "multiplexing" interfaces

Avoid:
```
// Provided by the browser to the renderer to support
```

```
// persisting window.localStorage.
interface DomStorageService {
  // `origin` is the origin of the Document.
  // Bad because a malicious sender can lie about its origin!
  SetData(url.mojom.Origin origin, string data);
};
```

Prefer:
```
// Provided by the browser to the renderer to support
// persisting window.localStorage. Each instance is
// 1:1 with a blink::Document.
interface DomStorageService {
  SetData(string data);
};
```
With the preferred pattern, a malicious caller cannot spoof its origin. Instead, since the IPC passes over a per-Document interface, the browser process can take advantage of the fact that it already knows the origin of the document associated with that instance of the interface. When handling IPCs, the browser process should implicitly use that trusted origin maintained by itself, minimizing the need to rely on information from an untrustworthy sender.

Another common sign of a multiplexing interface is an interface that accepts

Use interface lifetime to implement resource acquisition is initialization (RAII) semantics

Avoid:
```
interface RendererKeepAlive {
  IncrementCount();
  DecrementCount();
};
```

Prefer:
```
// Empty interface: closing the message pipe will decrement the keep alive count.
interface RendererKeepAliveHandle {};
interface RendererKeepAlive {
  RegisterKeepAlive(pending_receiver<RendererKeepAliveHandle> keep_alive);
};
```

With the preferred pattern, a malicious renderer cannot cause the keepalive count to go negative or wrap around.

Scope capabilities and privileges with separate interfaces

This is an extension of the previous two points, but is security critical. Avoid:
```
// Allow the browser process to monitor and modify network requests.
// Primarily used to implement the legacy webRequest extension API.
interface NetworkRequestMutator {
  WillRequestURL(URLRequestParams params);
};
interface NetworkService {
  SetMutator(pending_remote<NetworkRequestMutator> mutator);
  CreateURLLoader(pending_receiver<URLLoader> loader);
};
```

Prefer:
```
// Allow the browser process to monitor and modify network requests.
// Primarily used to implement the legacy webRequest extension API.
interface NetworkRequestMutator {
  WillRequestURL(URLRequestParams params);
};
interface PrivilegedNetworkService {
  SetMutator(pending_remote<NetworkRequestMutator> mutator);
}
// Usable by any process that makes network requests, including
// the renderer process.
interface NetworkService {
  CreateURLLoader(pending_receiver<URLLoader> loader);
};
```
The idea is to allow any caller access to `NetworkService`, while restricting `PrivilegedNetworkService` access to only to a few (or even a single) trustworthy caller.

Evaluating .mojom changes

All changes must have non-test usage

This includes:

- adding a new Mojo interface
- adding a new Mojo method
- adding/changing a parameter of a Mojo method
- adding/changing a field of a Mojo struct/union
- et cetera

Rationale

- It is easier to assess potential security impacts from an IPC change when you can see the end-to-end context in the same CL. At least one sandbox escape bug has been missed because it was unclear that a newly implemented interface would be exposed to the renderer.
- An IPC review covers more than just the .mojom file[1]. The security impact of an IPC change may depend on implementation and usage details that cannot be determined just by looking at the .mojom files.

Exceptions

This requirement is in tension with the philosophy of sending out smaller and more reviewable CLs. Some ways to mitigate this:

- When a new interface requires a lot of boilerplate: create an initial CL with a new Mojo interface with no methods and include the plumbing for binding and requesting the interface in that CL. Followup CLs will add new methods to the interface with the corresponding use and implementation.
- When a new type (structs, unions, and enums) requires a lot of boilerplate (e.g. for defining typemaps): it is acceptable to create an initial CL with just the types, as long as there is a dependent, followup CL. You should be included as a reviewer for both CLs.
- If there is no better alternative: work with the CL author to determine an appropriate way to break the change into smaller CLs. You should be included as a reviewer for any of the smaller CLs which implement or use the Mojo changes.

Comments

Comments should help the reader understand the purpose of something, and not just repeat what it is. Concretely, avoid comments like this:

```
// Interface for observing a media player.
interface MediaPlayerObserver {
  // An error occurred with `message`.
```

[1] For the purposes of Code-Review+1, //ipc/SECURITY_OWNERS are only marked as OWNERS for *.mojom and *_mojom_traits.* files due to the limitations of enforcing the IPC review requirement within the OWNERS framework. In principle, any change that affects IPC should go through IPC review, even if no *.mojom files are changed, e.g. when introducing a new implementation of a Mojo interface.

```
  OnError(string message);
};
```

Which simply restate the name of the interface/method and do not add any information. Instead, prefer:

```
// Used by the browser process to observe playback events in a media
// element (e.g. <audio>, <video>) in the renderer process.
interface MediaPlayerObserver {
  // An error occurred during playback (e.g. the media stream data does not decode).
  // `message` is a human-readable string intended for display to the user.
  OnError(string message);
};
```

The specifics of what comments should include will be covered in more detail in the next few sections.

Interfaces

Interfaces should have comments that describe:

- The purpose of the interface
- Entity or entities (C++ or otherwise) the interface has a relationship with, e.g. many interfaces for the web platform are 1:1 with a blink::Document.
- An overview of usage, e.g. if a state machine is part of the interface, is there an order that methods must be called in?
- What process (or processes) the interface is implemented in. While Mojo notionally allows an interface to live in any process, in practice, the majority of interface implementations live in one process.
- What process (or processes) use the interface.

Methods

Methods must have a comment that describe:

- The purpose of the method.
- The arguments to the method. Reading these comments should be enough to give you a general idea of what values may be supplied as inputs or returned as outputs.
- Any preconditions or postconditions, e.g. if a method needs to be called as part of a set sequence (but also see minimize “statefulness” within an interface).
    - While reading, work to understand what guarantees that these conditions are true—or if nothing guarantees that those conditions will be true, what happens when they are not? Missing cases in comments often match up with cases that are not robustly handled in code.

Look for parameters that are unnecessary or can be eliminated. Examples:

- a per-Document interface should not need to pass the origin or URL of the document over Mojo IPC; the browser process and the renderer process both already know the origin and the URL of the document.
- a per-renderer-process interface that needs to pass a document origin or URL over IPC can often be refactored to be per-Document or per-ExecutionContext. After that, passing the origin or URL over IPC becomes unnecessary.

Overspecified or redundant parameters can lead to security bugs where values can be internally inconsistent, and different parts of the code have different ideas about which parameter is the canonical one. For example, avoid defining a method like this:

```
ProcessData(string buffer, uint32 buffer_size);
```

As a string, `buffer` already has an implicit size, but `buffer_size` is its own redundant parameter—what happens if it does not match `buffer`'s implicit size?

Structs and Unions

Note: much of this section also applies to a method's input and output parameters, which are essentially a struct as well.

A struct or union must have a comment that explains what it represents. Individual fields should have clarifying comments as needed.

Many structs and unions are mapped to a corresponding C++ type (typemapping); in those cases, it is acceptable to reference the documentation for the C++type, though it is still preferable to include a high-level summary in the .mojom file itself.

Things to watch for:

- Types with complex invariants are likely candidates for C++ typemapping (see below). This allows the invariants to be validated in a single place at the Mojo IPC boundary rather than requiring all users of the type to validate the data. It is also more robust; it is mistakenly to omit validation checks when they are centralized into a typemap.
- Avoid redundant fields. Similar to method parameters,
- Use the type system to enforce invariants. Fields that are mutually exclusive are often candidates for grouping into a Mojo union. Parallel arrays of data can be rearranged as an array of structs, e.g. avoid:

  ```
  struct BufferData {
    array<int32> ids;
```

```
  array<mojo_base.mojom.ReadOnlySharedMemoryRegion> regions;
};
```

Prefer:

```
struct IdAndRegion {
  int32 id;
  mojo_base.mojom.ReadOnlySharedMemoryRegion>region;
};
```

And then passing around `array<IdAndRegion>` instead.

Types

Scrutinize strings, array<uint8>, et cetera

Strings are useful but often not the best / strictest representation. Look for:

- Strings where only a small set of values are legal. Consider using an enum instead.
- Strings / byte buffers that need to be parsed in some way.
    - If this parsing is happening in a trustworthy process, the parsing must not violate the rule of two.
    - Consider if the sender can parse the data into a more structured format before sending it, e.g. if the string is JSON, the sender can parse it and send it over IPC as mojo_base.mojom.Value.
    - Exception: serializing / deserializing untrustworthy protobufs is allowed, even in a trustworthy process. Protobufs sent in this way should use the mojo_base::ProtoWrapper (best) or mojo_base.mojom.ByteString type and have a comment that references the underlying protobuf type.

Prefer specific types over general types

- Origins should use url.mojom.Origin, not url.mojom.Url (and certainly not string).
- URLs should use url.mojom.URL, not string. Constructing an origin from a URL is a lossy operation and may not always give the correct result (e.g. for about:blank or for a sandboxed iframe).
- File paths should use mojo_base.mojom.FilePath, not string. But:
    - File names should use mojo_base.mojom.BaseFileName, not mojo_base.mojom.FilePath. mojo_base.mojom.BaseFileName protects against path traversal attacks (e.g. `../../../../../etc/passwd`) and performs other platform-specific checks (e.g. disallowing `NUL` on Windows).
    - It is often much better to pass a file handle as a mojo_base.mojom.ReadOnlyFile or mojo_base.mojom.File instead, as this allows for a stricter sandbox policy. Do not use handle<platform> or int32 to pass file handles.

- Time units should use mojo_base.mojom.Time, mojo_base.mojom.TimeDelta, or mojo_base.mojom.TimeTicks as appropriate. `int32 milliseconds` or `double seconds` should be avoided.

Enforce invariants with types

- Indexes, sizes, and offsets into a buffer should generally be unsigned types.
    - Corollary: if an index, size, or offset needs to be signed, there must be a comment that explains what it means when the value is negative.
- Arrays that must always be a fixed size must be specified as such, e.g. `array<int32, 3>` for an array that must always contain three int32 elements.

Avoid handle<T>

handle<message_pipe>, handle<platform>, et cetera should be rarely needed.

- Message pipe handles are usually end points for an interface; use `pending_receiver<Interface>` or `pending_remote<Interface>` instead.
- Raw platform handles can often be wrapped by pre-existing //base constructs, such as:
    - mojo_base.mojom.ReadOnlyFile
    - mojo_base.mojom.File
    - mojo_base.mojom.ReadOnlySharedMemoryRegion
    - mojo_base.mojom.WritableSharedMemoryRegion
    - mojo_base.mojom.UnsafeSharedMemoryRegion
    - et cetera

Nullability

If a parameter or struct field is nullable, there should be a comment that describes when it may be omitted and what it means if the value is not present. Otherwise, do not use nullable. For example, do not use a nullable string (i.e. `string?`) if the null value and the empty string have the same meaning.

Versioned Mojo interfaces are an exception to this rule: new parameters and new fields in versioned structs must be marked as nullable. An explicit comment should still be included if nullable has an additional meaning beyond “the sender is using an older definition of the method or struct”.

Enums

Enums must have a comment that explains what it represents. Individual enumerator definitions should have clarifying comments as needed.

Discourage:

- Use of an enum for a bitfield, i.e. adding the [Extensible] attribute to the enum to (mostly) disable validation. Instead, use a struct of bools or an unsigned integer with defined constants for the various flags.
- Placeholder / sentinel / otherwise unused enumerator values. One common place this comes up is recording enums in histograms. Mojo automatically emits kMinValue / kMaxValue C++ enumerators, so avoid:
  ```
  // in .mojom
  enum Colors { kRed, kGreen, kBlue, kCount, };

  // in C++
  UMA_HISTOGRAM_ENUMERATION(
      "Background.Color", color, mojom::Colors::kCount);
  ```
  Prefer:
  ```
  // in .mojom
  enum Colors { kRed, kGreen, kBlue, };

  // in C++
  // UMA_HISTOGRAM_ENUMERATION() automatically infers the range from
  // the autogenerated mojom::Colors::kMaxValue when the last argument
  // is omitted.
  UMA_HISTOGRAM_ENUMERATION("Background.Color", color);
  ```

Evaluating C++ changes

Though Chrome is actively investing in improving the safety of C++, it is not possible to enumerate all the ways that malicious or unexpected input can trigger undefined behavior. This section includes broad categories and specific examples to look for, but should not be considered exhaustive.

Data Validation

- Data validation should happen as soon as possible after data is received.
- A trustworthy process must be careful to validate any data that is passed over IPC from an untrustworthy process. Data validation is highly context-specific, but some ideas for what to watch for:
  - If an untrustworthy process claims to act on behalf of a particular origin or URL, is that something that can happen legitimately?
  - Arithmetic calculations should be done using the helpers in //base/numerics/checked_math.h to avoid arithmetic overflow/underflow bugs.

    - An index, offset, or size should not be assumed to be valid; the trustworthy process must validate that the value is in the appropriate range.
- <something about redundant stuff here>
- Avoid re-validating invariants that are guaranteed by the Mojo layer. For example, a `pending_receiver<Interface>` or `pending_remote<Interface>` are not nullable unless explicitly specified as `pending_receiver<Interface>?` or `pending_remote<Interface>?`. Mojo itself already enforces null correctness. Do not attempt to gracefully handle a non-nullable parameter being null–though either a DCHECK() or CHECK() is fine.
- Avoid using mojo::TypeConverter; use typemapping instead.

Validation Antipatterns

Look for validation checks that do not belong in the interface implementation. One common way this can manifest is when the same check is duplicated into multiple method implementations. Some examples:

- Checks can be done before binding an interface: an interface that only the primary main frame may use should reject attempts to bind the interface for non-primary main frames by calling ReportBadMessage(), rather than each method manually checking that the RenderFrameHost is the primary main frame.
- Checks that arguments/structs are internally consistent: for example, consider a method like:
  ```
  ProcessImage(gfx.mojom.Size size, array<uint8> pixels);
  ```
  Any method implementation working with a size and a pixels needs to validate that the pixels buffer is actually large enough to back an image of `size`. It would be better to bundle the two parameters together into one struct:
  ```
  struct Image {
    gfx.mojom.Size size;
    array<uint8> pixels;
  };
  ```
  and use a typemap to consistently perform this validation everywhere the Image struct is used.

Memory Unsafety

C++ is not a memory-safe language, so IPC handling must not allow an untrustworthy process to violate memory safety in a more trustworthy process. Things to look for:

- DCHECKs: in general, avoid DCHECK. DCHECKs are compiled out of official builds and will provide no security for end users. DCHECK() should only be used to assert internal

class invariants; if a remote caller can affect the invariant, the invariant is no longer an internal invariant.

- CHECKs: can an untrustworthy process violate these conditions somehow? If so, failing the check will crash a privileged process on a bad IPC. Instead, kill the sender process with `ReportBadMessage()`.
- raw pointers to other objects: what guarantees that the pointed-to objects always outlive `this`? If a Mojo interface implementation contains raw pointer fields, each field should have a comment documenting the lifetime of the pointed-to objects.
- Use of base::Unretained() in conjunction with IPC should be annotated to describe safety. This commonly comes up in conjunction with Mojo in two situations:
    - Setting a disconnect handler:
      remote_->set_disconnect_handler(base::BindOnce(
        &MyClass::OnRemoteEndDisconnected,
        base::Unretained(this));
    - The reply callback for an async call:
      remote_->DoSomething(
        arguments,
        base::BindOnce(
          &MyClass::OnDidSomething,
          base::Unretained(this)));
- mojo::MakeSelfOwnedReceiver<MojoInterfaceType>(...) is convenient but prone to potential use-after-free errors, especially when the self-owned receiver has raw pointers to other objects, or when other objects have raw pointers to the the self-owned receiver. The lifetime of the instantiated interface implementation only ends when the message pipe for the interface is closed. If the caller lives in an untrustworthy process, an attacker may be able to extend or shorten the lifetime of the self-owned receiver in surprising ways and trigger use-after-frees through dangling pointers.
- Rule of 2 violations, e.g. parsing untrustworthy input in the browser process. Instead, have the untrustworthy caller parse it into a more structured form, e.g. the network service model of pre-parsing certain headers into the ParsedHeaders struct for the browser process to safely consume.

Shared Memory

Be cautious of shared memory usage, especially if an untrustworthy process is writing to a shared memory mapping. Shared memory is prone to time-of-check to time-of-use issue:
```
// sender is untrustworthy
auto mapping = region.Map();
DataHeader* header = mapping.GetMemoryAs<DataHeader>();
if (!ValidateMappingIsLargeEnoughForData(region, header))
  return;
// Dangerous: an untrustworthy sender can change `entry_count` after
```

```
// the validation check above.
for (size_t i = 0; i < header->entry_count; ++i) {
  Process(header->entry[i]);
}
```

Protecting against this requires making a copy of `entry_count` and caching that locally:

```
// sender is untrustworthy
auto mapping = region.Map();
DataHeader* header = mapping.GetMemoryAs<DataHeader>();
const size_t entry_count = header->entry_count;
if (!ValidateMappingIsLargeEnoughForData(region, entry_count))
  return;
// Safe: `entry_count` is a local copy and an untrustworthy sender
// cannot change it in the middle of processing.
for (size_t i = 0; i < entry_count; ++i) {
  Process(header->entry[i]);
}
```

Shared memory is also prone to info leaks in padding bytes. memcpy()—or even just normal field assignment—can lead padding bytes. Due to these types of subtle bugs, consider if the use of shared memory is needed at all.

Note: BigBuffer can use shared memory for large messages.

Reject Invalid Input

Invalid inputs should be gracefully handled in a trustworthy process: invalid input should not trigger a crash or another more severe bug. However, a trustworthy process should also disown the (potentially malicious) untrustworthy process using mojo::AssociatedReceiver<T>::ReportBadMessage() / mojo::Receiver<T>::ReportBadMessage() (preferred) or mojo::ReportBadMessage() (deprecated).

This will close the Mojo message pipe associated with the interface, preventing further communication. In addition, calling this in the browser process when handling an IPC from the renderer process will terminate the renderer process for sending a bad IPC.

Note: after calling `ReportBadMessage()`, remember to abort any additional IPC handling (e.g. early return and propagate the failure all the way up the stack). A surprisingly common bug is reporting a bad message, but then erroneously continuing onwards: https://crbug.com/1285384

Typemapping

Typemapping allows a generated Mojo struct, union, or enum to be mapped to a corresponding C++ type. This should be used when it is difficult to enforce invariants solely through the type system. Note that typemapping is not a universal panacea for enforcing security invariants either though. Typemapping happens in a context-free environment, so a check like “does this frame have permission to access the example.com origin” is usually not possible to implement in a typemap.

Things to look for:

- Do not memcpy() a C++ struct/class to/from a byte buffer to serialize or deserialize it. memcpy() is a source of potential info leaks when used to copy structs or classes, as it also copies the internal padding bytes. Instead, declare an explicit Mojo struct with identical fields and explicitly map between the C++ fields and the Mojo fields.
- If a typemapped type T has a built-in nullable state[2], ensure that it is typemapped with `nullable_is_same_type`. Otherwise, Mojo will wrap the type in absl::optional<T> when it is specified as nullable in the Mojo IDL. This can lead to confusion, e.g. consider the typemap:
  ```
  {
    mojom = "mojo_base.mojom.RefCountedString"
    cpp = "::scoped_refptr<::base::RefCountedString>"
  }
  ```
  A Mojo method defined as:
  ```
  interface Processor {
    Process(mojo_base.mojom.RefCountString? string);
  };
  ```
  Will generate the following C++ signature:
  ```
  void Processor::Process(
      absl::optional<scoped_refptr<base::RefCountedString>> string) { … }
  ```
  absl::optional<scoped_refptr<base::RefCountedString>> has two layers of nullability: does code also need to check that the inner scoped_refptr is not null when the outer absl::optional is non-null?

  Instead, with `nullable_is_same_type = true`:

[2] Whether or not T should have a built-in nullable state is a complex design question that involves different tradeoffs. But a Mojo typemap should aim to minimize confusion and maximize consistency with any pre-existing semantics.

```
{
  mojom = "mojo_base.mojom.RefCountedString"
  cpp = "::scoped_refptr<::base::RefCountedString>"
  nullable_is_same_type = true
}
```
The generated signature becomes:
```
void Processor::Process(scoped_refptr<base::RefCountedString> string) { … }
```

Disconnect Handlers

Disconnect handlers should be used when:

- graceful error recovery is possible if a remote process crashes.
- the other end of the message pipe is explicitly closed to signal an event

The call to `set_disconnect_handler()` must be annotated with a comment that explains the purpose of the disconnect handler, i.e. one of the above reasons. If closing the message pipe is used as an explicit signal, this convention must also be documented in the interface's comment.

Other usage should be scrutinized, e.g. the renderer process cannot really recover from a browser process crash, so using a disconnect handler to "handle" that is pointless and misleading.

Document-Scoped Interfaces

Many Mojo interfaces exist to implement web platform functionality. Where possible, these interfaces should not be process-wide and should be strongly associated with a document or worker via BrowserInterfaceBroker.

Do not use mojo::MakeSelfOwnedReceiver<MojoInterfaceType> to instantiate document-scoped interfaces. The implementation may outlive the lifetime of the document and lead to security bugs, such as performing security checks using the wrong origin.

Instead, use content::DocumentService<MojoInterfaceImpl> (if there can be multiple instances of the interface per document) or content::DocumentUserData<MojoInterfaceImpl> (if there can only be one instance of the interface per document) instead, which safely encapsulates the lifetime of a document.

Blink-specific Quirks

- Any class the derives from blink::GarbageCollected<T> should use blink::HeapMojoReceiver<T>, blink::HeapMojoRemote<T>, et cetera, instead of mojo::Receiver<T>, mojo::Remote<T>, et cetera. **Failure to use the Blink types can lead to memory safety bugs with Blink and v8's garbage collected (Oilpan) types.**
- Typemapping is not compatible with Oilpan types. It is common and expected to write mojo::TypeConverter<MojoType, CppType> specializations to aid in converting between the generated Mojo types and the C++ types in Blink. That being said, type conversions should still signal errors in conversion. The best way to do that depends on what is being returned:
    - Mojo structs and unions can be returned by MojoTypePtr (a type alias for mojo::StructPtr<MojoType> or mojo::InlinedStructPtr<MojoType>), returning a null MojoTypePtr to signal error.
    - Garbage-collected types can be returned by raw pointer, returning nullptr to signal failure.
    - POD types such as enums can be returned with an absl::optional<Enum> wrapper, using absl::nullopt to signal failure.

Evaluating Java changes

TODO(someone): Add more Java-specific tips.

- Java code runs in the browser process, and all considerations about validating untrustworthy data apply.
- Java code can smuggle pointers to C++ objects inside `long` variables, typically with a prefix of `mNative` or `native`. Be cautious when these are used to call back into C++ code; if the safety of the C++ object lifetimes is unclear, ask the CL owner to clarify and document why it is safe.
- Unlike C++, Java does not support typemapping. It is more common to see handwritten code for manually marshalling to and from the generated Mojo Java types. This is not ideal but currently unavoidable. Encourage such conversion code to live in a location that can be shared, and ensure that it rejects invalid inputs.

Evaluating JavaScript changes

The majority of Mojo JavaScript changes involve either changes for WebUIs or Web Tests.

Like Java, JavaScript bindings suffer from the problem of not supporting typemapping, so it is common to see code have to deal with the messy internals of things like mojo_base.mojom.Time.

In general, prefer stricter typing in the Mojo definitions over ease of implementation in JavaScript: the stricter typing provides safety and readability benefits for the C++ handlers in the browser process responsible for managing communication with the page in the renderer.

WebUI

WebUI is a method for building UI surfaces in Chrome using HTML + JS, such as chrome://settings. Like regular web pages, WebUI pages are still hosted in a sandboxed renderer process. Unlike regular web pages, WebUI pages can utilize APIs that are not exposed to normal web renderer processes.

These privileged APIs are exposed to the WebUI page using one of two mechanisms:

- Using the legacy `chrome.send()` JS method to send JSON objects to the browser which are translated into base::Value objects. Browser □ renderer communication via chrome.send() is technically IPC but is does not go through IPC review.
- Using Mojo with the JS bindings. Each WebUI page defines its own set of Mojo interfaces for IPC which go through the normal IPC review process.

Trustworthy vs. Untrustworthy WebUIs

There are two types of WebUIs, trustworthy and untrustworthy. Chrome uses the same-origin policy to create a strong security boundary between the two types of WebUIs: trustworthy WebUIs use the chrome:// scheme, and untrustworthy WebUIs use the chrome-untrusted:// scheme. While trustworthy WebUIs may only be loaded in the outermost main frame, an untrustworthy WebUI may be implemented as a standalone page, or embedded as an iframe within a trustworthy WebUI.

Trustworthy WebUIs (chrome://)

Trustworthy WebUIs are considered an extension of the browser process, so they often possess powerful capabilities. For example, chrome://downloads can request the OS shell to open a downloaded file—even potentially dangerous files like binary executables. Beyond any WebUI-specific interfaces exposed to a given WebUI page, a WebUI page hosted in chrome:// can currently also request **any** renderer-exposed interface. **This may change in the future.**

Interfaces for trustworthy WebUIs should be bound in `PopulateChromeWebUIFrameBinders()`. Using the standard binding pattern enforces two important checks:

- That the requesting renderer process is allowed to host trustworthy WebUI.
- That the requesting page has access to the Mojo interface, e.g. only chrome://downloads is allowed to request downloads.mojom.PageHandlerFactory.

Important: despite the name, trustworthy WebUIs are still considered less trustworthy than the browser process. Browser process WebUI handlers **must** validate the data received

from a WebUI renderer process; malformed inputs must not result in a browser process crash or worse. Similarly, the browser process must exercise caution in what data it exposes to the WebUI renderer process: among other things, the browser process must not send the address of browser process objects to the renderer process.

Untrustworthy WebUIs (chrome-untrusted://)

The `chrome-untrusted://` scheme indicates that the WebUI page handles potentially untrustworthy content (often user-generated content), e.g. rendering an image, parsing a PDF, et cetera.

The `chrome-untrusted://` scheme does not indicate that the WebUI page is designed to perform malicious actions, or that users should not trust it. Instead, it is a signal to readers and reviewers that the WebUI renderer process should be assumed to be compromised, much like an ordinary renderer process.

Unlike trustworthy WebUI pages, which may request any renderer-exposed interface through `Mojo.bindInterface()`, an untrustworthy WebUIs is restricted to requesting its single bespoke Mojo interface registered in `PopulateChromeWebUIFrameInterfaceBrokers()`. Often, this interface is used as a top-level broker interface to expose additional interfaces to the untrustworthy WebUI.

Communication between WebUIs

Trustworthy WebUIs can embed untrustworthy WebUIs in an <iframe> to process potentially untrustworthy content. Communication between a trustworthy and untrustworthy WebUI should use Mojo rather than `window.postMessage()`. An exception to this guideline is permitted when transferring JavaScript objects that cannot easily be sent over Mojo e.g. `FileSystemFileHandle`.

Just like any other IPC between a trustworthy and untrustworthy process, reviewers should ensure the IPC makes sense. For example, consider a hypothetical chrome:// WebUI that is granted unrestricted Bluetooth access. Simply proxying this Bluetooth access through to a chrome-untrusted:// WebUI would provide an untrustworthy process with an excessively powerful capability and defeat the point of the original security boundary. A much safer alternative would be allowing the untrustworthy WebUI to request the trustworthy WebUI to show a Bluetooth device chooser dialog on its behalf, and then pass the selected device back to the untrustworthy WebUI.

Common Patterns

The most common pattern in MojoJS WebUI is to provide a pair of PageHandler/Page Mojo interfaces.

By convention, PageHandler is implemented in the browser to provide capabilities to the WebUI renderer, e.g. allowing the renderer to query the browser process about a user pref. Page provides communication in the opposite direction and is implemented by the WebUI page in the renderer to receive notifications from the browser process, e.g. the browser process notifying the chrome://bluetooth-internals page about a change in Bluetooth status.

Additionally, many WebUIs implement a PageHandlerFactory interface for simultaneous bootstrapping of the PageHandler and Page interfaces.

```
interface PageHandlerFactory {
  CreatePageHandler(pending_remote<Page> page,
                    pending_receiver<PageHandler> handler);
};

interface PageHandler {
  GetDownloads(array<string> search_terms);
  // ...
};

interface Page {
  RemoveItem(int32 index);
};
```

Web Tests

Web Tests can access some special Mojo interfaces for automating Web APIs (e.g PermissionAutomation) or for providing fake implementations (e.g. FakeBluetoothChooser). Binder callbacks for web test-only interfaces should only be registered in:

- WebTestContentBrowserClient::RegisterBrowserInterfaceBindersForFrame() for per-frame test interfaces.
- WebTestContentBrowserClient::ExposeInterfacesToRenderer() for per-process test interfaces.

These test-only interfaces often have powerful capabilities and are not safe to expose to normal web pages. For safety, access to these interfaces is guarded at both compile time and at run time—the interfaces are only compiled as part of content_shell, and they are only dynamically registered if `--run-web-tests` is passed on the command line.
[TODO: Any other aspect that should be included?]

Appendix

What makes a process (more) trustworthy?

The Chrome browser uses process separation as a security boundary. Any given IPC spans at most[3] two processes. Often[4], one of these processes is considered more trustworthy and the other process is considered less trustworthy.

In general, a process is considered less trustworthy when it handles untrusted input (e.g. parsing and running JavaScript from a page) and it violates the rule of 2.

In general:

- The browser process is the most trustworthy process. It manages all other processes and should be considered the authoritative source of truth.
- The renderer process is the least trustworthy process. It is directly responsible for processing untrusted input, e.g. loading pages and executing JavaScript.
- Most other processes lie somewhere in between. The GPU process accepts drawing commands from the renderer (which is untrustworthy), so it is sandboxed–at the same time, the sandbox is weaker than the renderer process sandbox, due to GPU driver requirements.
- A utility process has variable trustworthiness since it can host many different types of services. The service sandbox that is active for a utility process is a gauge of how trustworthy a given utility process is–e.g. on Windows, some shell operations are proxied to a utility process that is unsandboxed, since the goal is to limit the damage a buggy third-party DLL that triggers crashes can cause. In other cases, the utility process exists solely to handle untrusted content, e.g. the DataDecoder service.

Missing Things

- Interface arity and mojo::Receiver vs mojo::ReceiverSet. LaCrOS has a tendency to use ReceiverSet.
- Type converters. And mention other forms type converters might take, and that those, barring some really exceptional reason, should be typemaps. They are more efficient and require (overall) less boilerplate.
- Size calculations for buffers should be scrutinized for overflow
- If every method of an interface passes an “ID”, that is a strong sign that the interface is being multiplexed.
- Add a section about binders and common binder locations.

[3] It is possible to use Mojo IPC to make a call within one process, but a single Mojo IPC is always from exactly one endpoint to exactly one other endpoint: there is no broadcast functionality built into Mojo.
[4] One common exception to this is Chrome OS, where both sides might be considered trustworthy and/or be privileged.

- Add section about using EnableIf in “All changes must have non-test usage”
- Cover when test-only APIs *are* acceptable.
- Document when writable files can be passed to an unprivileged process (e.g. is it ok to pass to the renderer? What about a utility process?)
- Encourage typemapping for bitsets
- Adding something about preferring unions (and eventually expected) for returning results that can fail

Examples of Things Gone Awry

Mojo validation confusion

Mojom interface definitions contain information about data types. In the case of an enum the Mojom even includes the valid range of values an integer can take. Generated bindings then enforce that the sender supplied values that fall within the valid range.

Although Mojo's built-in message validation determines whether a message matches its specification, there is still room for accidents.

crbug.com/1406115 is an example where Mojo validation wasn't quite sufficient by itself or was perhaps confusing. Here's a pseudocode summary of what happened.

A Mojom enum specifies the range of expected values for a message field.

```
C/C++
/* .mojom */
enum my_enum {
      INVALID = -1;
      A;
      B;
      C;
      D;
      MY_MAX_VALUE;
};
```

Then the generated C++ bindings had a validation check like this:

```
C/C++
  static bool IsKnownValue(int32_t value) {
```

```
    switch (value) {
      case -1:
      case 0:
      case 1:
      case 2:
      case 3:
      case 4:
        return true;
    }
    return false;
}
static bool Validate(int32_t value) {
    return IsKnowValue(value);
}
```

Good so far, but in the receiving code we saw something like this:

C/C++

```
/* A C++ enum to match the Mojom enum. */
enum my_enum : int8_t {
  INVALID = -1,
  A,
  B,
  C,
  D,
  MAX_NUM_VALES
};

/* Fixed size allocation based on the max expected value. */
SomeType array[MAX_NUM_VALUES]; // [Note 1]

/* Use the received value as an index into the array */
array[value_from_mojo_after_validation]; // [Note 2]
```

Mojo validation ensures that the received value is between -1 and 4, and the array allocation at Note 1 creates space for 4 entries. However, index -1 or 4 would be an out of bounds access. In the case of crbug.com/1406115, the -1 case was handled safely, but the latter was not.

During IPC security reviews, take careful notice when

- a value received from Mojo is used as an index into an array. If such a design is unavoidable then strongly encourage author to use safer array-like data structure like base::span.
- an enum is C++ is with a kMax value.