Rusty QR Code Generator

This Document is Public

Short link: go/rusty-qr-code-generator
Author: lukasza@chromium.org
April 2023

Reviewer	danakj	Approved
Reviewer	Adam Langley	Approved
Reviewer	Peter Williamson	Approved
Reviewer	Elly Fong-Jones	Approved

One-page overview

Summary

The Chrome Security team is actively working on adding a production Rust toolchain to the Chromium build system. The Rusty QR Code Generator project will be used for verifying that the Rust toolchain works well when actually shipping a third-party Rust crate in the Chrome binary - this project will be the first to ship and use such a crate. (A chrome://crash/rust URL handler will ship earlier and will be the first project to ship Rust-compiled code in Chromium, but it will not use any third-party Rust crates - see https://crbug.com/1368726.)

The QR Code Generation has been chosen as the initial, pilot usage of Rust, because:
- It is used on the majority of Chrome platforms where the Rust toolchain needs to work.
- It requires only a fairly simple and narrow FFI between C++ and Rust
- It can be easily reversed if it causes issues (since it is a relatively small feature and it is not under pressure from any product goals).

We expect that switching to a third-party Rust crate will also lead to minor improvements in Chromium (e.g. code simplification, performance improvements), but such gains are only a secondary motivation for this project. In particular these secondary benefits haven't been evaluated from the perspective of business needs that third-party Rust crates are expected to meet (at this point during Chromium Rust experiments piloting the Rust toolchain provides sufficient justification for the project).

The QR generator in Chromium started life as a way to generate linking QR codes for passkeys. In that context it only processed trustworthy data generated inside Chromium and so ran synchronously. Later Chromium added the ability to share pages via QR codes (try clicking the share icon in the omnibar), but URLs are arbitrary inputs and the QR generator isn't trivial. So, by the rule of two, it was moved into a utility process. But that makes it asynchronous and slow, which complicates the lifetimes of objects involved in the UI and adds a noticeable UI delay on slower machines. The Rusty QR Code generation projects plans to make QR generation synchronous, fast and safe. We plan to do that by delegating QR generation into a 3rd-party Rust crate (rather than using Chromium's current C++ implementation).

Platforms

Mac, Windows, Linux, Chrome OS, Android (justification: sharing the URL of the current page via QR is available on all of these platforms; WebAuthn QR generation is used on all desktop platforms).

~~iOS~~ (not sure if that would be affected by changes in chrome/browser/… - does it use Chromium's QR code for webauthn and/or url sharing? It seems that for sharing on iOS a different QR generator is used.)

~~Fuchsia, Android WebView, WebLayer~~ (justification: QR codes are currently only generated for display in the browser UI, and these have no browser UI)

~~Android CCT~~ (justification: Adam Langley points out that WebAuthn QR generation is desktop only; Peter Williamson has checked that we intentionally don't make the sharing dialog available for CCTs).

Team

Łukasz Anforowicz (from Chromium Security > Chrome Memory Safety > Cr2O3 / Rust-in-Chromium team) is responsible for the design, implementation, and launch of the project.

Long-term owner of the project is Peter Williamson (from the Chrome With Friends team)

Bug

Implementation-tracking bug: https://issues.chromium.org/issues/40263739
(previously tracked as https://crbug.com/1431991)

Launch-tracking bug (Google-internal): https://launch.corp.google.com/launch/4248932

Code affected

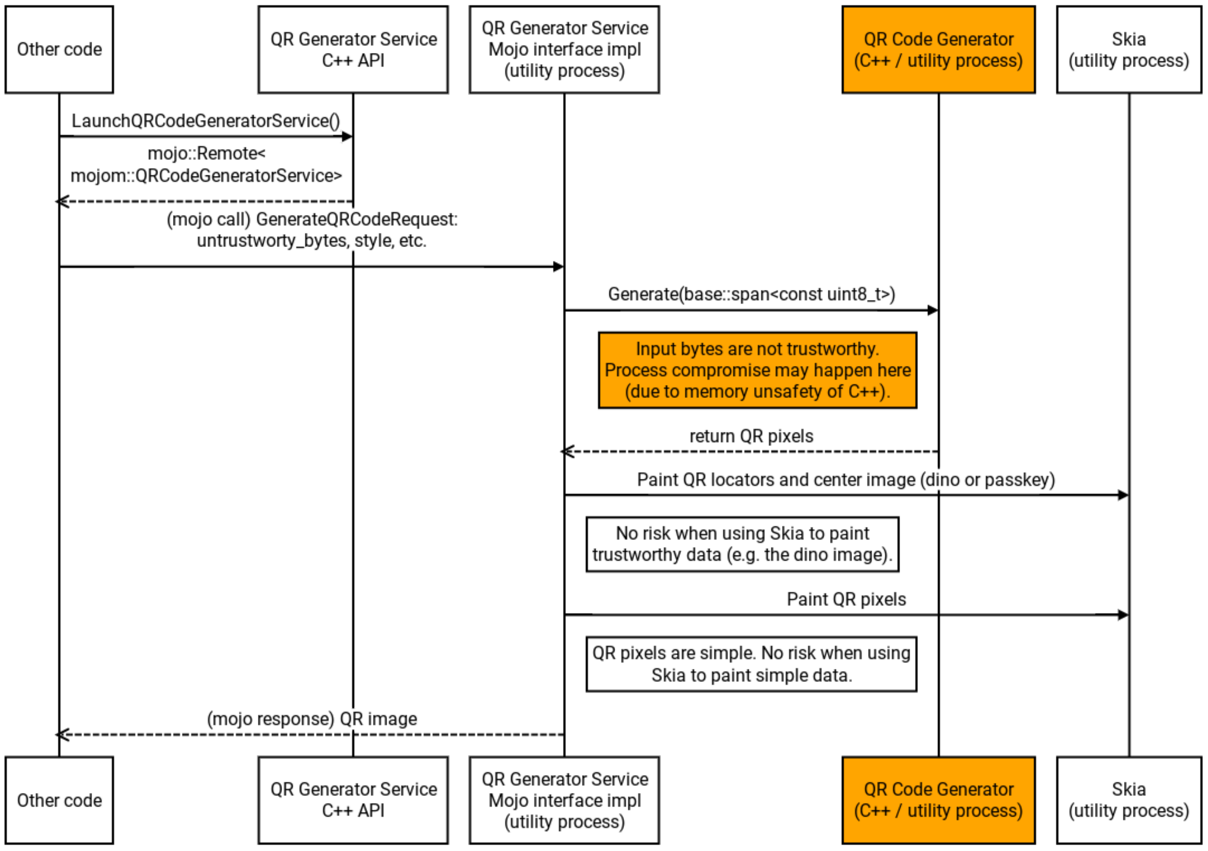
- QRCodeGenerator - //components/qr_code_generator/…
    - This is a process-agnostic, somewhat low-level code that can translate a sequence of bytes into QR pixels
    - This project plans to *replace* this code with a 3rd-party Rust crate
    - This code is used from Ash OOBE for nearby and cellular connectivity
- QRCodeGeneratorService - //chrome/services/qrcode_generator/…
    - This provides an API that the Browser process uses to generate QR bitmaps. The implementation runs in a utility process and:
        - Uses QRCodeGenerator (the item above) to translate bytes into QR pixels
        - Translates QR pixels into a Skia image
    - This project plans to *modify* this code:
        - Expose an in-process, synchronous C++ API instead of the current mojom API
        - Use the Rust crate that replaces C++ QRCodeGenerator.
    - This code is used for webauthn on desktop, Ash RMA generation, and for URL sharing on desktop and on Android

---

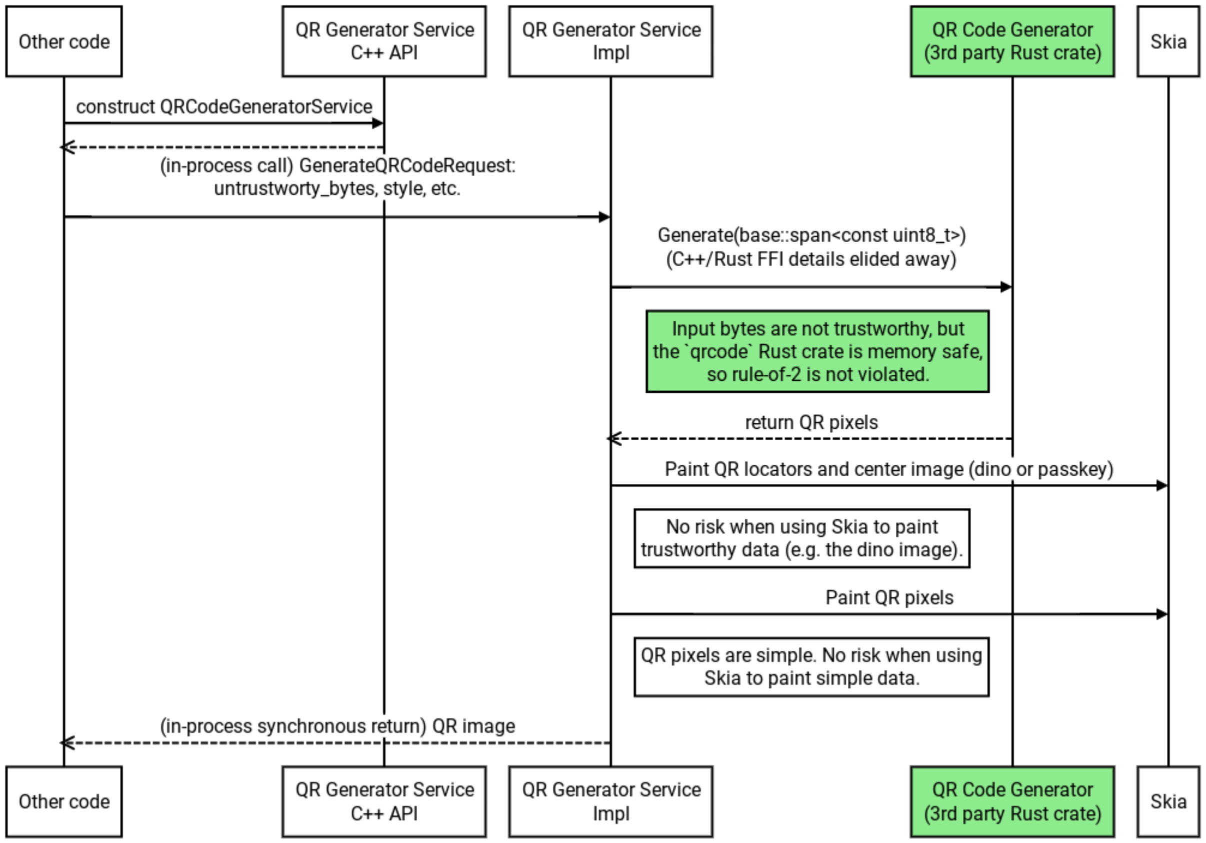
Design

High-level overview of the planned changes

Current implementation (diagram source can be found here):


Other code
QR Generator Service C++ API
QR Generator Service Mojo interface impl (utility process)
QR Code Generator (C++ / utility process)
Skia (utility process)
LaunchQRCodeGeneratorService()
mojo::Remote< mojom::QRCodeGeneratorService>
(mojo call) GenerateQRCodeRequest: untrustworty_bytes, style, etc.
Generate(base::span<const uint8_t>)
Input bytes are not trustworthy. Process compromise may happen here (due to memory unsafety of C++).
return QR pixels
Paint QR locators and center image (dino or passkey)
No risk when using Skia to paint trustworthy data (e.g. the dino image).
Paint QR pixels
QR pixels are simple. No risk when using Skia to paint simple data.
(mojo response) QR image
Other code
QR Generator Service C++ API
QR Generator Service Mojo interface impl (utility process)
QR Code Generator (C++ / utility process)
Skia (utility process)

Planned implementation (diagram source can be found here):


Other code
QR Generator Service C++ API
QR Generator Service Impl
QR Code Generator (3rd party Rust crate)
Skia
construct QRCodeGeneratorService
(in-process call) GenerateQRCodeRequest: untrustworty_bytes, style, etc.
Generate(base::span<const uint8_t>) (C++/Rust FFI details elided away)
Input bytes are not trustworthy, but the `qrcode` Rust crate is memory safe, so rule-of-2 is not violated.
return QR pixels
Paint QR locators and center image (dino or passkey)
No risk when using Skia to paint trustworthy data (e.g. the dino image).
Paint QR pixels
QR pixels are simple. No risk when using Skia to paint simple data.
(in-process synchronous return) QR image

Evaluation of available Rust crates

Chosen crate: qr_code

Based on the evaluation below, the project tentatively plans to use the `qr_code` crate.

Notes:

- As discussed in the section below, the segmentation approach is not optimal, but seems sufficient for Chromium scenarios.
- Security review of the `qr_code` crate can be found here: go/qr-code-chromium-security-review (Google-internal, sorry).

Overview of considered crates

- http://docs.rs/qrcode
  - Recent downloads: 246,569
  - Updated: 3 years ago
  - Notes:
    - Segmentation uses a simplistic greedy algorithm (also see here).

        - At the first glance the API should work okay with C++/Rust FFI provided by `cxx`
        - The maintainer is non-responsive (there are currently 7 open PRs with no reply from the maintainer)
- https://docs.rs/qrcodegen/1.8.0/qrcodegen/
    - Recent downloads: 68,481
    - Updated: 12 months ago
    - Notes:
        - No segmentation is implemented currently.
        - Author doesn't follow semver semantics (see the issue here).
        - At the first glance the API should work okay with C++/Rust FFI provided by `cxx`
- **https://crates.io/crates/qr_code [the chosen crate]**
    - Downloads all time: 23,598
    - Updated: 2.5 years ago
    - Notes:
        - This is a fork of `qrcode` (see kennytm/qrcode-rust/issues/51) that is more actively maintained.
        - Notable commits:
            - Removed dependencies / modules:
                - checked_int_cast dep (commit link)
                - image, svg, and render
            - Added #![forbid(unsafe_code)] (PR link)
- https://crates.io/crates/fast_qr
    - Recent downloads: 10,536
    - Updated: 18 days ago
    - Notes:
        - No segmentation is implemented currently.
- N/A - actual QR code generation is delegated to other crates
    - https://docs.rs/qrcode-generator => delegates to `qrcodegen`
    - https://crates.io/crates/qr2term => delegates to `qrcode`

Requirement: QR segmentation

The project requires that the generated QR codes use reasonable segmentation. This requirement has been initially pointed out by Adam Langley.

QR codes encode the input bytes into a number of segments. Each segment encodes a slice of input using a specific mode: numeric, alphanumeric, bytes, Kanji. Naive segmentation (e.g. encoding everything in "bytes" mode; or always using "numeric" mode for digits) may produce unnecessarily long QR codes.

Segmentation from the perspective of Chromium inputs

Note that the “alphanumeric” mode only covers upper-case letters. Case-insensitive URL elements (scheme, host name) are typically written as lowercase, and case-sensitive elements like paths commonly include lowercase. This means that URLs usually cannot be encoded as a single “alphanumeric” segment.

Chromium scenarios from segmentation perspective (AFAIK this covers all transitive users of //components/qr_code_generator/…):

- EID and FIDO URLs:
    - 2 callers:
        - cellular_setup/euicc.cc: "EID:" (alphanumeric) + 32 digits (numeric)
        - device/fido: “FIDO:/” + followed by digits (numeric)
    - For these scenarios the initial, naive segmentation (consecutive sequence of digits = “numeric” mode segment; an alphanumeric sequence => “alphanumeric” mode segment) is optimal.
- Arbitrary URLs:
    - 4 callers:
        - ash/login/oobe_quick_start: "https://signin.google/qs/…?key=…" (here - bytes, lowercase is *not* alphanumeric) + shared secret (base64 - probably bytes) + session id (also base64). Note that base64 may contain sequences of consecutive uppercase letters and digits (i.e. fit into an alphanumeric segment).
        - shimless_rma: url from a protobuf (probably bytes, because of lower-case)
        - chrome/browser/share: url from Java (probably bytes, because of lower-case)
        - chrome/browser/ui/views/qrcode_generator: url from NavigationEntry, from GetLastCommittedURL(), etc.
    - Initial, naive segmentation may produce suboptimal results - merging adjacent segments might be required to minimize the length of the QR code (e.g. to avoid representing a sequence of N alternating digit/uppercase-letter like “1A2BC3…” with 2N segments: N “numeric” segments interleaved by N “alphanumeric” segments).
    - In practice, optimization opportunities might have limited impact, because uppercase characters are fairly rare in URLs.

Algorithms in current Chromium code and in considered Rust crates

The current C++ code greedily coalesces adjacent segments in 2 passes: first pass merges “numeric” and “alphanumeric” segments and the second pass also considers “bytes” segments (adjacent segments are merged if it results in a local improvement). The

C++ code doesn't consider or produce Kanji segments. The algorithm has O(N^2) time complexity.

The `qrcode` crate implements a single-pass greedy coalescing algorithm (also see here) that merges adjacent segments if it results in a local improvement. The algorithm runs in O(N) time (merging is done in an Iterator::next implementation, and unlike Chromium's implementation doesn't remove elements from a vector, making it more efficient in memory bandwidth as well). FWIW, a doc comment says that the algorithm "does not use Annex J from the ISO standard".

The following example shows where **the 2-pass coalescing algorithm produces a better result than a single-pass coalescing algorithm** (such as used by the `qrcode` crate): 20 '#' characters followed by 20 alternating letter / digit pairs (i.e. "####################A1B2C3D4E5F6G7H8I9J0").

- The optimal segmentation is a "bytes" segment followed by an "alphanumeric" segment.
- The single-pass greedy algorithm will repeatedly consider it beneficial to merge the initial "bytes" segment with each of the subsequent single-character "numeric" or "alphanumeric" segments.

This example is not representative of Chromium use cases:

- `qrcore`'s algorithm will produce optimal segmentation for Chromium's EID: and FIDO: URLs. This was verified for sample EID: and FIDO: URLs - `qrcode` produced reasonable segmentation for them (a short Alphanumeric segment followed by a longer Numeric segment). See here for a live, in-browser demo.
- Segmentation optimization opportunities have limited impact on other scenarios where arbitrary URLs are processed, because upper-case letters are relatively rare in URLs.

Project Nayuki provides a QR code generation library in multiple languages (including the `qrcodegen` crate). One of the **project's pages describes a dynamic programming approach for generating optimal segmentation**. Unfortunately, this algorithm is only available in the Java version of the library (as documented here). In particular, **the `qrcodegen` crate does *not* implement any segmentation optimization** - the implementation of `QrSegment::make_segments` always returns a vector of length 0 or 1 (the documentation of this method is somewhat misleading saying: "may use various segment modes and switch modes to optimize the length of the bit stream").

Side-note: The author of this design doc (Łukasz Anforowicz) believes that the segmentation problem can be seen as the problem of finding the shortest path in a graph (where graph nodes represent positions in-between bytes of input, and each graph edge decides the mode to be used to encode the previous byte). The dynamic programming approach can be seen as a particular implementation or variant of the Dijkstra algorithm (where nodes and edges are considered in a kind of "hard-coded" order, rather than in an order dictated by a priority queue).

The `fast_qr` crate doesn't implement any optimization of segmentation. A single segment is used for the whole QR code. See QRBuilder::build, QRCode::new: (single mode: `let mode = encode::best_encoding(input)`), create_matrix (single mode received as input), encode (single mode received as input - single call to encode_numeric / encode_alphanumeric / encode_byte).

Additional optimization opportunities

Chromium's algorithm and the `qr_code` algorithm only merge 2 segments at a time, and only if the merge is immediately/locally beneficial. This strategy produces suboptimal results when merging 3 or more segments is needed to realize some gains - this can happen if a merge results in an increase in bits-per-char that needs to be offset by savings from dropping *more* than 1 segment header.

On the first 2880 characters of base64 input from https://cryptopals.com/sets/1/challenges/6 this results in the following difference:

- Shortest-path algorithm: 23034 bits (see https://github.com/RCasatta/qr_code/pull/23)
- Single segment of bytes: 23060 bits
- `qr_code` algorithm: 23356 bits (101.4% of shortest-path-based segmentation)
- Chromium algorithm: 23666 bits (102.7% of shortest-path-based segmentation)

A hostile input (160 x [8 bytes followed by 8 alphanumerics]) can be constructed where merging of 2 consecutive segments is never beneficial, but a single segment of bytes is optimal:

- Single segment of bytes: 20500
- Original segmentation: 23200 (113.1% of optimal segmentation)

Requirement: API that can replace Chromium's QRCodeGenerator

The project requires that the 3rd party Rust crate provides an API that can 1) replace Chromium's QRCodeGenerator, 2) use C++/Rust FFI tools available in Chromium (primarily `cxx`, although in theory manually-crafted FFI and `cbindgen` are also available).

On Apr 10, 2023 the C++ library in Chromium provides the following API (see here):

- **Input: a sequence of arbitrary bytes**: `base::span<const uint8_t>`. In practice the input is always UTF-8 (see the Chromium scenarios section above), but the C++ API doesn't enforce this (switching to `std::string` could weakly signal the intent, but wouldn't help with enforcement).
- **Config**:
    - Minimal QR code version

        - Most non-test callers don't specify this optional argument
        - One caller passes 5 (here) saying: "The QR version (i.e. size) must be >= 5 because otherwise the dino painted over the middle covers too much of the code to be decodable."
        - Note that the C++ implementation supports only QR version up to 12 (i.e. it is capable of encoding up to ~300 arbitrary bytes). The `qr_code` Rust crate supports up to QR version 40 (i.e. it can encode up to ~2800 arbitrary bytes).
    - QR code mask (used to avoid long swaths of all-white or all-black pixels)
        - Seems unused (see the removal CL here - https://crrev.com/c/4420267)
    - Error correction level is not controlled by the caller.
        - A comment says that "all versions currently use error correction at level M".
          (Interestingly `QrCode::new` in the `qr_code` crate also defaults to `M`.)
        - Wikipedia describes the following levels: L (7% can be restored), M (15%), Q(25%), H (30%). Requesting higher error correction levels may be potentially desirable, but is outside the scope of this project.
- **Output** (see here):
    - raw QR pixels / modules: `base::span<uint8_t> data`
        - Each byte indicates either "dark" or "light" pixel based on least-significant bit (see here). Other bits are masked out and ignored (service layer - see here and here; lower layer - see here and here).
        - This always contains `qr_size * qr_size` pixels. The rendering code will ignore locator areas.
    - QR size: `int qr_size` (this is the width [and also the height] of the QR code [unit = QR modules / pixels])
    - Error is indicated by an empty `data` (see here). Note that INPUT_TOO_LONG is handled and may be returned at the mojom layer.

The `qrcode` crate provides the following API:
- Input:
    - Accepts arbitrary bytes (takes `AsRef<[u8]>` as input - e.g. see QrCode::new).
- Config:
    - Either automatically chooses the best version/size of the output (QrCode::new or QrCode::with_error_correction_level) or takes a *specific* (rather than *minimal*) version to use (QrCode::with_version). This is okay - if the former method returns a version smaller than the requested minimal version, then we can retry by asking for exactly the minimal version.
- Output:
    - QrError can report the following errors:
        - Input data too long
        - Invalid version / error correction level (N/A - we won't specify)

        - Unsupported character set / invalid character / invalid ECS (N/A - can only happen when using low-level APIs)
    - QrCode::into_colors takes `self` and returns `Vec<Color>`.
        - This covers the locator areas as well (like C++ code) - see here (the quiet zones surround the QR code, but `i` is incremented for each "internal" pixel, including the locator areas).
        - C++/Rust FFI bindings provided by `cxx` seem able to support this API via `rust::Vec<T>` (since T is an opaque Rust type, rather than an opaque C++ type - the latter is unsupported). We would just need to provide some accessor methods - e.g. `Color::is_dark()`, or convert to `Vec<u8>` in the FFI/glue code (the latter is probably preferable because of performance considerations).
    - QrCode::width takes `&self` and returns `usize`. Note that this is the width (height is the same), not the total number of pixels.

The `qrcodegen` crate provides the following API:
- QrCode::encode_binary accepts arbitrary bytes (takes `&[u8]`)
- `minversion` is accepted by `QrCode::encode_segments`
- The only possible error is `DataTooLong`
- Size is provided via `QrCode::size`
- There is no direct way to get a vector of all the pixels - an intermediate layer needs to repeatedly call QrCode::get_module(&self, x, y) to get a `bool` at the given coordinates.

Proposed public API changes

//components/qr_code_generator

Summary of the existing public API at components/qr_code_generator/qr_code_generator.h:
    - // member (i.e. non-`static`) function:
      absl::optional<QRCodeGenerator::GeneratedCode>
      QRCodeGenerator::Generate(
         base::span<const uint8_t> in,
         absl::optional<int> min_version,
      ~~   absl::optional<uint8_t> mask~~) // Unused - removal in https://crrev.com/c/4420267
    - struct GeneratedCode {
         base::span<uint8_t> data; // 0-length upon errors, points at `QRCodeGenerator::d_`
         int qr_size = 0; // Invariant: data.size() == qr_size * qr_size
      }

Proposed API changes:
- Remove the unused `mask` parameter (motivation: simplification)
- Change `GeneratedCode::data` to `std::vector<uint8_t>`
  - Motivation: avoiding non-owning pointers/references across the FFI boundary
  - Motivation: avoiding invalidating `base::span` after a 2nd call to `Generate`

Optional API changes (nice-to-haves, not required for project success):
- Communicate errors via `base::expected<GenerateQRCodeResponse , QRCodeGeneratorError>` rather than via `absl::optional`
- Reduce unused public API surface (e.g. `QRCodeGenerator::SegmentType`, `VersionClass`, `V5`, etc. can be made `private` [or comments may be added that they are private for unit tests])
- Change `qr_size` to `size_t`

//chrome/services/qrcode_generator

Summary of the existing public API at chrome/services/qrcode_generator/public:
- C++:
  - mojo::Remote<mojom::QRCodeGeneratorService> LaunchQRCodeGeneratorService();
- Mojo:
  - enum QRCodeGeneratorError
  - enum ModuleStyle
  - enum LocatorStyle
  - enum CenterImage (only Dino and Passkey are used in practice it seems)
  - struct GenerateQRCodeRequest {
      string data;
      bool should_render; // Whether to populate `GenerateQRCodeResponse::bitmap`
      ModuleStyle render_module_style;
      LocatorStyle render_locator_style;
      CenterImage center_image;
    }
  - struct GenerateQRCodeResponse {
      QRCodeGeneratorError error_code;
      skia.mojom.BitmapN32? Bitmap; // null on error, or if `!should_render`
      array<uint8> data; // data from the //components/qr_code_generator layer
      gfx.mojom.Size data_size; // data from the //components/qr_code_generator layer
    }
  - [ServiceSandbox=sandbox.mojom.Sandbox.kService]
    interface QRCodeGeneratorService {

```
  GenerateQRCode(GenerateQRCodeRequest request)
      => (GenerateQRCodeResponse response);
};
```

Proposed API changes:

- Replace mojo calls with C++ calls:
    - Proposed API:

        -
          ```
          class QrCodeGeneratorService {
           public:
            // Default-constructible, movable, non-copyable.
            QrCodeGeneratorService();
            ~QrCodeGeneratorService();

            void Generate(
                mojom::GenerateQRCodeRequestPtr request,
                base::OnceCallback<void(mojom::GenerateQRCodeResponsePtr)>
          callback);
          }
          ```
    - Motivation:
        - In the current, C++/mojo implementation the proposed changes hide (i.e. encapsulate away): `mojo::Remote` and mojo calls (allowing to replace them with in-process calls based on a base::Feature).
        - Controlling the mojo calls and callbacks means that the mojo response can first be processed by QrCodeGeneratorService - this will help measure and record the new UMA metric planned for the project.
        - Having a QrCodeGeneratorService class (instead of making `Generate` a top-level free function) means that we can keep a long-lived `mojo::Remote`. This seems somewhat desirable (as opposed to getting a `mojo::Remote` within the `Generate` function and establishing this mojo connection on demand - e.g. every time a QR code generator happens) as it allows the Remote to become optional in the future, with the QrCodeGeneratorService being the only code aware of the choice between local vs remote generation. Currently users of the old API stash the `mojo::Remote` in the following fields (and it seems reasonable to assume that some of these fields can be long-lived):
            - ChromeShimlessRmaDelegate::qrcode_service_remote_
            - QRCodeGenerationRequest::remote_
            - QRCodeGeneratorBubble::qr_code_service_remote_
            - AuthenticatorQRViewCentered::qr_code_service_remote_
    - Other notes:

  - The proposed API still uses`mojom`-defined `GenerateQRCodeRequest` and `GenerateQRCodeResponse`. This is a bit icky in a C++ API, but facilitates code reuse in the period when C++/mojom and C++/Rust implementations coexist.
  - The Rust implementation may call `callback` synchronously.

Follow-up API changes if the Rusty QR Code Generator project is successful:

- Move definitions of `enum QRCodeGeneratorError`, `enum ModuleStyle`, `enum LocatorStyle`, `enum CenterImage`, `struct GenerateQRCodeRequest`, `struct GenerateQRCodeResponse` from `mojom` to `.h`
- Eliminate the GenerateQrCodeRequest and pass its values as parameters to Generate(...) directly?
- Remove the Callback parameter from Generate(...) and return base::expected<QrCodeResponse, QrCodeGeneratorError>. QrCodeGeneratorError is removed from QrCodeResponse.
- Remove the `QrCodeGeneratorService` class and refactor the `Generate` member function into a top-level free function (e.g. named `GenerateQrCode`).

Follow-up Views work

- The views QR bubble code requires careful C++ lifetime manipulation due to the asynchronous QR code generation, and this complexity should be removed. This item is likely owned by Peter Williamson. Specifically, QRCodeGeneratorBubble relies on the fact that ~QRCodeGeneratorBubble tears down the Mojo remote, which cancels any pending requests and avoids a UAF when the request completes. QRCodeGeneratorBubble also has a whole partially-initialized starting state because of the QR code generation request being asynchronous, which should be removed.

FFI solution

For calling Rust from C++, the project plans to use the `cxx` crate (which already is integrated into Chromium's build system and GN templates). This necessitates writing a thin layer of Rust code that presents a simplified, FFI-friendly API on top of the API exposed by the `qr_code` crate:

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        fn generate_qr_code_using_rust(
            data: &[u8],
            min_version: i16,  // plain integer instead of `Option<Version>`;  -1 = no min
version
            out_pixels: Pin<&mut CxxVector<u8>>,  // plain `u8` instead of `Color` enum
            out_qr_size: &mut usize,
        ) -> bool;  // `bool` result + `out_...` parameters are used instead of `Result<T, E>`
    }
}
```

Additionally (as pointed above in the "Requirement: API that can replace Chromium's QRCodeGenerator" section) the `qr_code` doesn't directly support a `min_version` parameter - this will be implemented as (calls into the `qr_code` crate are in **bold**):

```
fn generate(data: &[u8], min_version: Option<i16>) -> Result<QrCode, QrError> {
    let mut qr_code = QrCode::new(data)?;

    let actual_version = match qr_code.version() {
        Version::Micro(_) => panic!("QrCode::new should not generate micro QR codes"),
        Version::Normal(actual_version) => actual_version,
    };

    match min_version {
        None => (),
        Some(min_version) if actual_version >= min_version => (),
        Some(min_version) => {
            // If `actual_version` < `min_version`, then re-encode using `min_version`
            qr_code = QrCode::with_version(data, Version::Normal(min_version),
EcLevel::M)?;
        }
    }

    Ok(qr_code)
}
```

Some additional notes about the FFI experience have been gathered in a document here (Google-internal, sorry).

Metrics

New metrics: Sharing.QRCodeGeneration.Duration...

There is no metric that measures the time it takes to generate a QR code. Existing metrics include:

- metadata/sharing/histograms.xml metrics (not related to QR codes)
- PageActionController.QRCodeGenerator.Icon.CTR in metadata/page/histograms.xml
- Mobile.Share.QRCodeImage.Actions in metadata/mobile/histograms.xml
- Network.Cellular.ESim.InstallViaQrCode... in metadata/network/histograms.xml

Therefore this project plans to introduce a new metric:
Sharing.QrCodeGeneration.Duration:

- The "Sharing" part is a bit inaccurate. "Sharing" is one feature area that depends on QR code generation, but QRCodeGeneratorService is also used in other scenarios and we plan to also cover these other scenarios in our measurements (example: chrome/browser/ash/shimless_rma).
- The new metric will measure the time from the point just before calling QRCodeGeneratorService::GenerateQRCode is called (this is a mojo call in the old browser process code, the project will replace this with an in-process call).
- The new metric will measure the time until the point when QRCodeGeneratorService (the C++ class executing in the Browser process, *not* QRCodeGeneratorServiceImpl that implements the mojo interface and executes in the utility process) calls the callback with GenerateQRCodeResponse (this is a mojo response callback in the old code, the project will keep using the callback for API compatibility but will call it synchronously).
- In the old code the bulk of performance impact is expected to come from having to launch a utility process (and some from the IPC overhead). In the new code the performance overhead is expected to be significantly lower (the FFI boundary is not expected to have a measurable impact).

Additionally more granular metrics will be introduced to measure:

- **Sharing.QRCodeGeneration.Duration.BytesToQrPixel2**: Only the bytes -> QR pixels time (i.e. the in-process time taken by `//components/qr_code_generator` and/or by `//third_party/rust/qr_code`).
- **Sharing.QRCodeGeneration.Duration.QrPixelsToQrImage2**: Only the QR pixels -> QR image time (i.e. the in-process time taken by the `//chrome/services/qrcode_generator` layer to paint QR pixels (and a Dino and/or a passkey) into a SkBitmap.

The expected impact of the project on the metrics above looks as follows:

- Avoiding a mojo call into a utility process should reduce Sharing.QrCodeGeneration.Duration
- Replacing `//components/qr_code_generator` by `//third_party/rust/qr_code` may regress Sharing.QRCodeGeneration.Duration.BytesToQrPixel and therefore may indirectly affect Sharing.QrCodeGeneration.Duration. We expect a regression in this metric, because the Rust library uses a different segmentation algorithm which (compared to the C++ version) takes more time to compute a more optimal segmentation. OTOH, this small regression should be paid back by bigger gains in the end-to-end Sharing.QrCodeGeneration.Duration metric.
- No impact on Sharing.QRCodeGeneration.Duration.QrPixelsToQrImage is expected

Success metrics

This project is successful if the newly introduced `Sharing.QrCodeGeneration.Duration` metric shows no regression. We aim for the low bar of "no regression", because of the improvements in code complexity (e.g. less first-code party code to maintain, no complexity associated with mojo and asynchronicity). OTOH, we actually expect that this metric should improve.

The project is not expected to directly impact the speed launch metrics.

Regression metrics

UMA metrics

We will monitor the following UMA metrics:

- Browser process memory
    - Memory.Browser.PrivateMemoryFootprint (**TODO**: there is no clear guidance on the UMAs to use - in particular the "new memory UMAs" doc linked from here seems abandoned? Still, the Canary+Dev experiments have highlighted Memory.Browser.PrivateMemoryFootprint as a potential risk, so let's monitor this metric)
- Browser process jankiness via:
    - Browser.MainThreadsCongestion
    - Browser.Responsiveness.JankyIntervalsPerThirtySeconds3
- Stability metrics

Build times

The initial usage of Rust in Chromium may prompt questions about the impact on build times. It is tricky to design an apples-to-apples comparison of Rust-vs-C++ build time, because

- The cost of dependencies of the C++ QR code is amortized across all of Chromium (e.g. the binary size impact and build-time impact of depending on the //base library and/or the C++ standard library), while the cost of dependencies of the Rust QR code will be largely specific to this project (because this will be one of the first projects that will use Rust in Chromium)
- Third-party Rust (and C++) libraries (such as `qr_code` used in this project) are expected to change infrequently - they should have no impact on incremental builds in day-to-day work of Chromium developers.
- C++ and Rust builds have inherently different performance characteristics:
    - Without C++ modules, C++ build times are a function of how many tokens are processed through all transitive includes, and our strategy to drive down that

number has been somewhat haphazard and working around the standard library. Rust build costs do not creep out beyond a crate in that same way.
    - Translation unit is typically bigger in Rust (crate) than in C++ (.cc file). In other words, `rustc` is typically invoked less often, but on bigger inputs than `clang`.

Because of the above the Rusty QR Code Generator project does *not* plan to explicitly measure or report build times. Measurements from the chrome://crash/rust project (the first project shipping Rust code in Chrome) can be found in a doc here (Google-internal, sorry).

We plan to give a heads-up to the ChOps team when landing a CL that adds the Rust crate to Chromium build on the CQ, so they can monitor if the load of bots or CQ times are impacted. We expect that the impact of the Rust code will be negligible compared to the overall Chromium build. The relevant CLs and Chrome versions are:

- 2023-06-16: r1159176: 116.0.5838.0: enable_rust_qr on Linux and Android:
- 2023-06-19: r1159552: 116.0.5843.0: revert of: enable_rust_qr on Linux and Android
- 2023-06-22: r1161494: 117.0.5849.0: reland of: enable_rust_qr on Linux and Android
- 2023-07-11: r1168771: 117.0.5884.0: enable_rust_qr on Windows and Fuchsia
- 2023-07-13: r1169965: 117.0.5888.0: enable_rust_qr on MacOS
- ????-??-??: r???: 117.0.???.0: enable_rust_qr on ChromeOS

Binary size

The initial usage of Rust in Chromium may prompt questions about the impact on the size of the Chrome binary. Comparing binary size is tricky - mostly because the cost of the base C++ libraries is amortized across multiple components, while the cost of the base Rust libraries will initially only support the QR crate (this is a bit similar to the disclaimer in the previous section about comparison of build times).

We plan to measure the binary size impact/delta (as reported by the Chromium Binary Size check on GerritSuperSize [see the example here from an unrelated CL]) for the following planned changes:

1) Impact of including the Rust crate in Chromium build
2) Impact of removing the C++ implementation and mojo bits

Note that the measurements will exclude some of the cost of shipping Rust from the chrome://crash/rust project (the first project shipping Rust code in Chrome) can be found in a doc here (Google-internal, sorry).

We will use the results to inform future decisions around Rust.

Binary size impact on Android

Impact of enabling Rusty QR Code Generation on Android: increase of 22kB. This is based on the official Chromium Binary Size CQ results from https://crrev.com/c/4509543 (OTOH note that the results fluctuate quite a bit because they depend not only on the Rust code but also on the effectiveness of LTO and other optimizations which depends on other code).

Impact of removing the C++ implementation and mojo bits: saving of 10kB. This is based on the official Chromium Binary Size CQ results from https://crrev.com/c/5091218.

Binary size impact on Windows

Impact of enabling Rusty QR Code Generation on Windows: https://crrev.com/c/4667673:

> Note that I wasn't able to build with `is_official_build = true` (trouble with "Rejected profile data for Call_ReceiverIsNullOrUndefined_Baseline_Compact due to function change"), therefore the results below don't include PGO and LTO:
>
> ```
> $ cat out/win/args.gn
> # is_official_build = true
> is_chrome_branded = true
> is_debug = false
> target_os = "win"
> use_goma = true
> ```
>
> (Also note that along the way we fixed additional exports from `chrome.dll` caused by https://crbug.com/1462356.)
>
> Before the CL:
>
> ```
> $ ls -l out/win/chrome.dll
> -rwxr-x--- 1 lukasza primarygroup 254435328 Jul 10 17:50
> out/win/chrome.dll
> ```
>
> After the CL:
>
> ```
> $ ls -l out/win/chrome.dll
> -rwxr-x--- 1 lukasza primarygroup 254486016 Jul 10 17:44
> out/win/chrome.dll
> ```
>
> Binary size delta (again, note that this is *without* `is_official_build`):
> 254486016 - 254435328 = 50688 bytes.

Impact of removing the C++ implementation and mojo bits: **TODO**

Experiments

This project plans to launch using a Finch experiment.

Experiment results can be found in go/rusty-qr-experiments (Google-internal, sorry…). As of Oct 11, 2023 Rusty QR Code Generator is

- enabled for 10% of the user population of the Stable release channel on Android, Linux, MacOS, and Windows in M117+
- enabled for 50% of the user population of the Beta channel on ChromeOS in M119+

Rollout plan

Rollout plan: standard experiment-controlled rollout using Finch.

Planned name of a future experiment: “RustyQrCodeGenerator”.

Given the low triggering percentage of the feature, a Chrome analyst suggests (see a Google-internal discussion here) to:

- Ask for approval directly for a higher percentage of Stable (say 10% or higher)
- Consider a careful launch with 28 days of data and 50% of Stable (to maximize the number of clients that participate in the experiment).

Core principle considerations

Everything we do should be aligned with and consider Chrome’s core principles.

Speed

We plan to monitor the speed launch metrics. We expect that the project will have a positive impact on the time needed to generate and display QR codes to the user. We expect that the project will have no or minimal impact on Browser jankiness (this risk comes from moving some processing from the utility process to the UI thread in the Browser process, though this same functionality was originally located in the browser process).

Simplicity

This project has no user-visible effects (other than the improved performance of generating and displaying QR codes in the browser UI).

Security

The input for QR code generation may be untrustworthy - for example the input may be the URL of the current page (such a URL may be controlled by an attacker that the page belongs to). An attacker may use a maliciously crafted input (e.g. an URL) to attempt to exploit bugs in the QR code generator (https://crbug.com/1177437 and https://crbug.com/1520419 are examples memory safety bugs in the C++ implementation). This can be done without requiring a navigation to the malicious URL.

The untrustworthy input is transformed into a QR image in two steps:

- Step 1: Translating untrustworthy input bytes (`base::span<uint8_t>`) into *simple* QR pixels: a size and a vector of size x size pixels (each pixel is a boolean value representing light or dark).
    - We claim that the QR pixels are “simple” and therefore are safe to process even if they originate from an untrustworthy input.
        - Current API: base::span<uint8_t> data
        - Proposed/new API: std::vector<uint8_t> data
    - **Side-note for security reviewer**: currently size has type `int`. To clarify intent it may be worth changing it to `size_t`, but this should have no security impact, because the size returned from the QR code generator is never negative and (in the current version of the QR standard) is at most 177 (size of Version 40 of QR codes).
- Step 2: Translating QR pixels (booleans) into QR image (SkBitmap).
    - The input for this step is
        - QR pixels from the previous step
        - Hardcoded (i.e. trustworthy) images of the Chrome dino and a passkey
    - The output from this step is a Skia image
        - Current and proposed API: skia.mojom.BitmapN32
        - Long-term API: non-mojo, C++ SkBitmap satisfying SkBitmapToN32OpaqueOrPremul() (with SkColorType == kN32_SkColorType, a platform-dependent RGBA vs BGRA ordering of 4 bytes-per-pixel, with row stride == row width * 4)

Currently both steps are implemented in C++. Both steps run in a utility process because of the Rule of Two (here we have: non-memory-safe language and untrustworthy input).

The Rust QR Generator project:

- Replaces step 1 with a call into a 3rd-party Rust crate
- Moves both steps out of the utility process / into the Browser process. This is okay because:
    - A memory-safe implementation of QR pixel generation in Rust means that the Rule of Two is not violated when processing the untrustworthy input.
    - Step 2 processes “simple” data

For a security review of the 3rd-party `qr_code` crate please see the doc here: go/qr-code-chromium-security-review (Google-internal, sorry)

Privacy considerations

This project is agnostic to the origin of the data. If QR codes encode privacy-sensitive data, then it is the responsibility of other features (i.e. features using the QR codes generator to encode such data) to consider the privacy impact and go through privacy review.

Testing plan

This section outlines tests that the Rust QR Code Generator project plans to depend on. tests:

Manual tests

Test team

Note to test reviewer: Let's discuss if the testcase below can be covered by the test team in the next few releases.

Manual end-to-end smoke tests might be desirable, especially if the "golden" tests planned below won't pan out. A manual smoke test is the most desirable:

- Steps to execute in Chrome:
    - Navigate somewhere (e.g. to https://example.com)
    - Generate a QR code
        - Desktop: focus the omnibox, click the qr code icon (kind of a square)
        - Android: click the triple-dot-menu, click "Share", click "QR Code"
        - (Maybe, depending if it uses this code) iOS: tap the share widget in the URL bar, drag the bottom panel up into view, tap "Create a QR Code"
- Other test steps:
    - Scan the QR code using a different device (another Android device is fine, but maybe using a non-Google device [e.g. an iPhone] would provide better signal/coverage).
    - Verify that the scanned QR code navigates to the right URL

Other notes:

- There is no platform-specific code under components/qr_code_generator nor chrome/services/qrcode_generator, but in the initial release it may be worth running the smoke test above on all platforms.
- TODO: long-url-test vs jankiness - Charlie Reis has kindly suggested using: https://md5calc.com/hash/crc32/This+is+a+really+long+string+to+encode.

TODO: can't use the long-url test before switching to Rust (C++ supports only shorter lengths)
TODO: do we need golden/pixeltest or just one-time/ad-hoc manual test?.

Dev team

We plan to verify that `EID:` and `FIDO:` examples produce reasonable segmentation. The manual verification steps would look more or less like this:

- Use Rust code path to produce a QR code for an arbitrarily chosen EID: or FIDO: URL.
- Open the generated QR image in a QR debugger (**TODO**: find a QR debugger that can show segmentation breakdown)

Automated tests

//components/qr_code_generator/qr_code_generator_unittest.cc

We plan to preserve the following unit tests (i.e. use them to test/cover both C++ and Rust implementations):

- QRCodeGenerator.Generate
- QRCodeGenerator.HugeInput

It is unclear if the QRCodeGenerator.ManySizes test is desirable in the long-term:

- The motivation for the test was a memory-safety bug in the C++ implementation: https://crbug.com/1177437
- This test can time out when run against the Rust implementation.
    - The test tries to encode increasingly long sequences of (non-alphanumeric) bytes. The C++ implementation can at most output QR version 12, which means the test covers lengths up to 288 bytes. The C++ implementation also defines `kMaxInputSize = 700` (the theoretical maximum for digits-only input, unlike in this unit test) which initially will be shared by the C++ and Rust implementation..
    - Amount of work needed to test the C++ implementation: 1 + 2 + … + 277 + 288 = 41616
      Work for Rust: 1 + 2 + … + 277 + 288 + 289 + … + 698 + 699 + 700 = 245,350 (C++ * 5.8)
    - Rust supports up to QR version 40, so in theory it can encode up to 2800 arbitrary bytes (or even larger digit-only inputs).

TODO: figure out if/how to preserve test coverage for the following aspects:

- components/qr_code_generator/qr_code_generator_unittest.cc
    - QRCodeGenerator.Segmentation - is testing below the level of the public API and therefore seems difficult to apply to the Rust version. Maybe the intent of the test can be replicated by asserting that the output is not-too-big (this is

challenging, because at the public API level we can only detect discreet bumps of the version/size number [width increases by 4 pixels for each QR “version” increase], so e.g. everything between [say] version 5 and 6 looks identical length-wise at this level]).

//chrome/services/qrcode_generator tests

AFAICT there is no coverage at this level before the Rust QR project (although individual users may provide some test coverage via their tests). Therefore this project plans to add the following tests:

- Smoke-tests via BrowserTests for each ModuleStyle, LocatorStyle, CenterImage verify that things look okay end-to-end (without actually inspecting the generated code and/or image in depth).
- Golden (pixel diff) tests
  - The output of the QR generator should be deterministic.
  - Rolling the QR generator library to a new version might force us to regenerate and re-test the goldens. This is probably okay if the number of goldens is limited.
  - Info about Chromium golden test framework: go/chrome-engprod-skia-gold (Google-internal, sorry). It seems that the `pixel_browser_tests` target is Windows-only (this should be ok - QR generation doesn’t have any platform-specific bits).
  - Proposed goldens:
    - Input - something easily verifiable manually - maybe a URL to https://example.com?
    - Golden #1 (see here):
      - Circles and rounded squares in (ModuleStyle, LocatorStyle)
      - Dyno in CenterImage
    - Golden #2 (see here):
      - Squares in (ModuleStyle, LocatorStyle)
      - Passkey in CenterImage

//third_party/rust/qr_code tests

`qr_code` crate’s native Rust tests will eventually execute on all appropriate Chromium bots and platforms. This will happen as part of the broader effort to run tests of *all* chromium/src/third_party/rust crates (this work is tracked under https://crbug.com/1304772 and we expect that it will complete in H2/2023). In the meantime we will rely on first-party test coverage outlined in the other sections above + on the crate’s tests being run by GitHub Continuous Integration.

Test coverage has been measured using the following commands:

```
$ CARGO_INCREMENTAL=0 RUSTFLAGS='-Cinstrument-coverage'
LLVM_PROFILE_FILE='cargo-test-%p-%m.profraw' cargo test
...

$ grcov . --binary-path ./target/debug/deps/ -s . -t html --branch --ignore-not-existing
--ignore '../*' --ignore "/*" -o target/coverage/html
...

$ DISPLAY=:20 google-chrome-stable target/coverage/html/index.html
```

The test coverage is 97.12% (3512/316) line coverage.

Followup work

Post-launch cleanup

If the launch is successful, then:

- DONE: https://crrev.com/c/5091218:
    - A significant part of the old code at components/qr_code_generator/... can be deleted. Some tests will be reused + minimal code will remain for calling into the Rust crate and making its output digestible by C++.
    - Mojo-specific code can be deleted
- DONE: https://crrev.com/c/5147349:
    - https://crbug.com/1334066: The old code does not support all QR versions up to 40 (see here). This limits `QRCodeGenerator::kMaxInputSize` (currently 700, while version 40 can support ~5600 characters at medium error-correction level). This limit can be raised when Rust is the only remaining version of the QR generator.
- DONE: The public C++ API of chrome/services/qrcode_generator/... can be tweaked to avoid having an asynchronous callback. See also the "Proposed public API changes" section above. Note that QR code generation can take 9ms (TODO: double-check this number with UMA) and therefore maybe it should be delegated to a thread pool?

If the launch is a failure, then:

- The project's code can be deleted with relative ease.

Opportunistic follow-ups

Notes about additional improvement opportunities if the project is successful:

- DONE(out-of-scope follow-up tracked in https://crbug.com/325664342): chrome/browser/ui/views/qrcode_generator/qrcode_generator_bubble.cc adds a

“quiet zone” around the generated QR image, but other users of chrome/services/qrcode_generator don’t.
    - Motivation:
        - This is an opportunity to unify the behavior and simplify code? (i.e. if the “quiet zone” was handled underneath chrome/services/qrcode_generator, then SkBitmap would be the only output / there would be no need to also output gfx::Size)
        - This is also an opportunity to prevent surprises - the generated image may or may *not* scan successfully without the quiet zone. And not every caller may know that the quiet zone is required.
    - Implementation notes:
        - This requires passing the background color (SkColor) as an argument of the //chrome/services/qrcode_generator layer. (And if background/quiet color, then we probably should also require explicitly specifying the foreground color.)
- DONE: Once (or maybe “if” :-)) the API is synchronous, we should switch to `base::expected<T, E>` (instead of an ad-hoc error-reporting convention used by //components/qr_code_generator)