Web Bluetooth Persistent Permissions

This document is public

Status: Work in Progress

Authors: odejesush@chromium.org

Reviewed by: reillyg@, ortuno@

Last Updated: 2020-05-13

One-page overview

Summary

Persist Web Bluetooth device permissions to allow sites to easily reconnect to permitted Web Bluetooth devices. Expose Web Bluetooth permissions to Site Settings and Page Info to allow users to have control over these permissions and allow users to grant a site permission to access a Bluetooth device only for the current browsing session.

Platforms

Linux, Mac, Windows, Chrome OS, Android
Android WebView will not be supported, since it requires WebView API changes to support Web Bluetooth.

Mailing List

web-bluetooth@chromium.org

Launch bug

https://crbug.com/974879

Code affected

Permissions, Bluetooth, Site Settings, WebUI, Page Info, Android Site Settings

Design

It is currently not possible for the Web Bluetooth API to enable pages to maintain a persistent connection to Bluetooth devices. The first issue is that the API does not provide a way to get a list of devices that a site has permission to access, so it has to prompt the user every time that it's loaded. The Bluetooth adapter logic also removes devices that haven't been seen in 3 minutes from the cache, so a new scan is needed in order to find these devices. Users are also unable to control permissions for Web Bluetooth because they are not exposed in settings UI. With persistent device permissions, it is important for the user to be able to reset permissions that have been granted to a site as well as be able to grant temporary device permissions.

To tackle the first issue, either the Permissions API Integration portion of the specification will be implemented or a getDevices() method will be added to the specification. Through these two APIs, a site can query the Bluetooth permissions and receive a list of Bluetooth devices that it already has access to.

Since Bluetooth devices are removed from cache after 3 minutes of not being detected, a way to create Bluetooth device objects using their address is needed. A new method will be added to the `BluetoothAdapter` that can return a `BluetoothDevice` given its address using platform specific APIs.

The last issue to resolve is allowing users to control Bluetooth permissions in settings UI and to be able to choose between granting permission once or indefinitely. The settings UI can be done by refactoring the Web Bluetooth permissions storage to use a class derived from `ChooserContextBase`, since the Site Settings and Page Info UIs already support displaying data from this class. This change would also make the Bluetooth permissions model and settings UI consistent with the ones for WebUSB, WebHID, and WebSerial. To allow the user to choose to grant a temporary or persistent permission, the chooser UI needs to be modified to display the option to do so. In the backend, the permissions storage should not persist permissions that are granted temporarily.

Detailed design

The first issue that needs to be solved is to give sites the ability to get a list of Bluetooth devices that they can use. This issue can be resolved with either the getDevices() API or the permission query algorithm. Since this change is public facing, it will be implemented behind a runtime enabled flag.

Retrieve Permitted Devices

WebBluetoothService::GetDevices()

The first task is to add an API to the `WebBluetoothService` that gets the permitted devices for the current site. The changes to the Web Bluetooth Mojo interface in web_bluetooth.mojom are as follows:

//third_party/blink/public/mojom/bluetooth/web_bluetooth.mojom

```
interface WebBluetoothService {
  GetDevices() => (Array<WebBluetoothDevice> devices);
};
```

The WebBluetoothService Mojo interface is implemented by WebBluetoothServiceImpl.

//content/browser/bluetooth/web_bluetooth_service_impl.h

```
class CONTENT_EXPORT WebBluetoothServiceImpl
    : public blink::mojom::WebBluetoothService,
      public WebContentsObserver,
      public BluetoothAdapter::Observer {
 private:
  // WebBluetoothService methods:
  // ...
  void GetDevices(GetDevicesCallback callback) override;
};
```

`WebBluetoothService` uses the `BluetoothAllowedDevices` class to store permissions. The `BluetoothAllowedDevicesMap` maintains a map of `BluetoothAllowedDevices` per origin. `BluetoothAllowedDevices` stores the permissions for Bluetooth devices for a given origin by associating the device's OS ID (MAC address for Windows, Linux, and Android and NSUUID for MacOS) to a generated `WebBluetoothDeviceId`. `BluetoothAllowedDevices` will need a new method that returns the `WebBluetoothDeviceId` and Bluetooth IDs pairs.

Then, GetDevices() can use device::BluetoothAdapter::GetDevice() for each Bluetooth ID to get the device::BluetoothDevice. This is needed in order to get the name for display to populate each WebBluetoothDevice mojo struct. Once the list of WebBluetoothDevice is ready, the callback can be run with the result.

This method will produce different results if BluetoothChooserContext is used. These differences are explained in the Deprecating BluetoothAllowedDevices section.

Bluetooth::getDevices() API

A simpler alternative to integrating with the Permission API is to add a getDevices() API to the Bluetooth interface, similar to the existing one for WebUSB.

```
[Exposed=Window, SecureContext]
interface Bluetooth : EventTarget {
  Promise<boolean> getAvailability();
  Promise<sequence<BluetoothDevice>> getDevices();
  attribute EventHandler onavailabilitychanged;
  [SameObject]
  readonly attribute BluetoothDevice? referringDevice;
  Promise<BluetoothDevice> requestDevice(optional RequestDeviceOptions options);
};
```

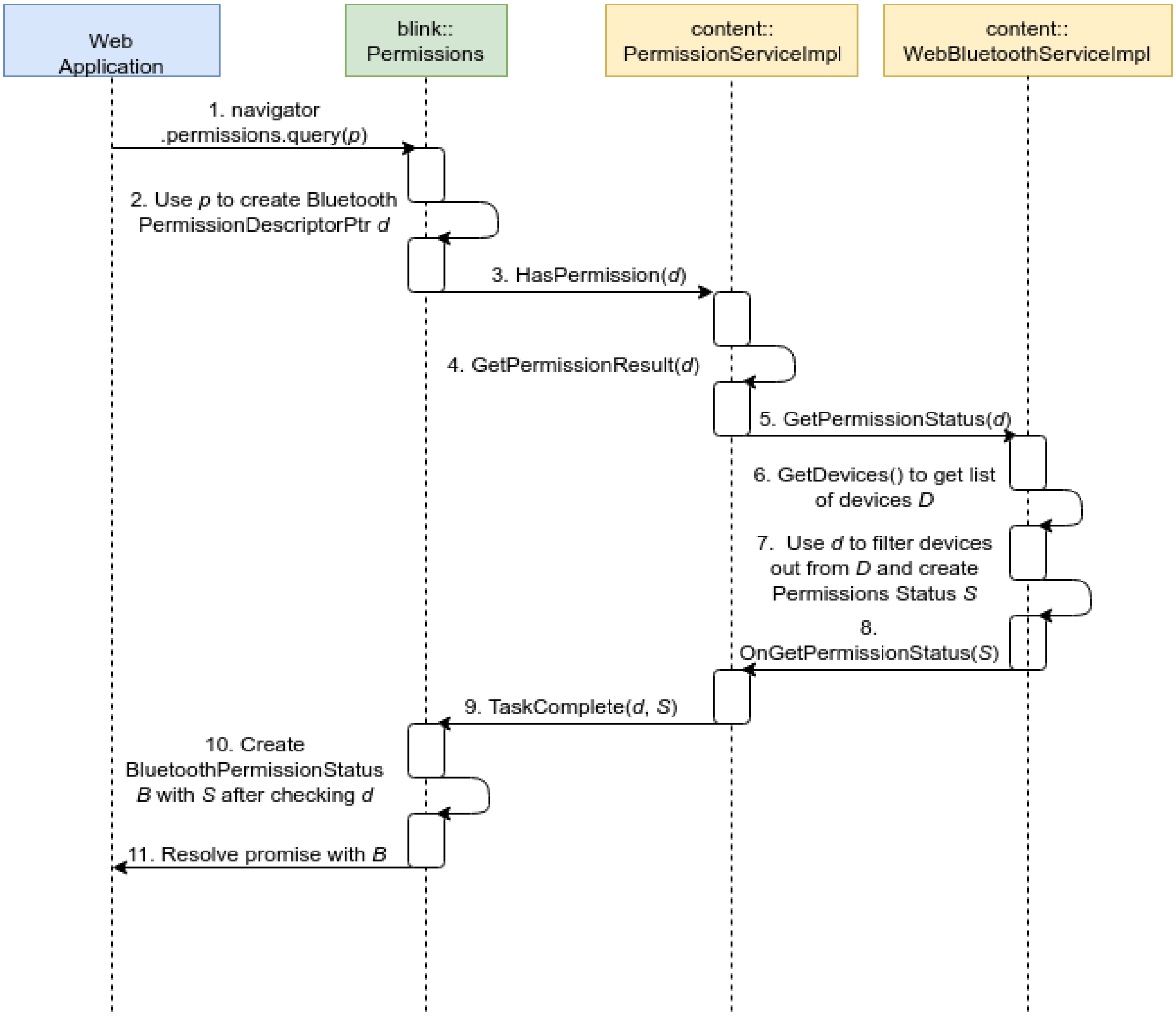
The Web Bluetooth spec will be updated with the algorithm for getDevices(). The devices returned by getDevices() may contain devices that are not currently in range and connected. The BluetoothDevice::watchAdvertisements() API can be used to detect when Bluetooth devices come into range of the Bluetooth radio. Then calling BluetoothRemoteGATTServer.connect() should resolve successfully if the device is able to be connected to.

To expose the new method, it needs to be added to the //third_party/blink/renderer/modules/bluetooth/bluetooth.* files. The IDL files needs to be modified to look like the code block above. The getDevices() method will perform the following steps:

1. Check that the current context is supported.
    a. If not, return a DOMException for a not supported error.
2. Check the the Web Bluetooth feature is enabled by feature policy.
    a. If not, return a DOMException for a feature policy blocked error.
3. Ensure that a WebBluetoothService connection exists or create one if it doesn't.
4. Call GetDevices() on the service with a callback to OnGetDevices() which will eventually resolve the promise with the results of GetDevices().

To use BluetoothChooserContext on a call to navigator.bluetooth.getDevices(), the BluetoothDelegate::HasDevicePermission() will need to be called during WebBluetoothServiceImpl::OnGetDeviceSuccess(). The ChromeBluetoothDelegate will forward the call to `BluetoothChooseContext::HasDevicePermission()`, which returns true if the given requesting and embedding origins have permission to access the Bluetooth device.

Permissions API



Web Application
blink:: Permissions
content:: PermissionServiceImpl
content:: WebBluetoothServiceImpl
1. navigator .permissions.query(p)
2. Use p to create Bluetooth PermissionDescriptorPtr d
3. HasPermission(d)
4. GetPermissionResult(d)
5. GetPermissionStatus(d)
6. GetDevices() to get list of devices D
7. Use d to filter devices out from D and create Permissions Status S
8. OnGetPermissionStatus(S)
9. TaskComplete(d, S)
10. Create BluetoothPermissionStatus B with S after checking d
11. Resolve promise with B

High-level overview of the Bluetooth Permissions API integration.

The Permissions API defines a common infrastructure that other Web APIs such as Web Bluetooth can use to interact with browser permissions. These interactions include the ability to query and request changes to the status of a given permission. The Web Bluetooth specification also defines how it integrates with the Permission API.

Chrome has an implementation for `navigator.permissions.request()`, but it is a non-standard API as it is not defined in the Permissions API specification. As a result, only `navigator.permissions.query()` will be implemented for Web Bluetooth. The method takes

`PermissionDescriptor` as a parameter and returns a `Promise` that resolves with a `PermissionStatus`. The specification for the Web Bluetooth API defines a `BluetoothPermissionDescriptor` and a `BluetoothPermissionResult` that extend `PermissionDescriptor` and `PermissionStatus` respectively. The Web Bluetooth API specification needs to be updated to rename `BluetoothPermissionResult` to `BluetoothPermissionStatus` in order for it to be consistent with the parent interface.

The `BluetoothPermissionDescriptor` enables the page to ask about permissions pertaining to Bluetooth devices and to filter out permissions that it is not interested in. A `deviceId` can be included in the descriptor to ask about the permission status for a specific device. A set of `BluetoothLEScanFilterInits` in the descriptor can further filter out devices that the page is not interested in. A set of `BluetoothServiceUUIDs` and a flag to accept all devices can also be included in the descriptor, but these are not used by the `query()` algorithm.

The `BluetoothPermissionStatus` contains the `PermissionState` for the Web Bluetooth API, an `onchange` event handler for the permission, and an array of permitted Bluetooth devices for the current origin. The `PermissionState` can be set to "denied" or "prompt" depending on whether the Web Bluetooth permission is set to "blocked" or "ask" by the user in Site Settings when this UI is implemented. The `onchange` event handler will be executed when the permission state changes (or when the permitted devices change?). Lastly, the `devices` array contains the permitted Bluetooth devices for the current origin.

To implement `query()`, the `blink::Permissions` class needs to be updated to handle a parameter containing a `BluetoothPermissionDescriptor` and return a promise that resolves with a `BluetoothPermissionStatus`. The new descriptor can be added as the following file:

//third_party/blink/renderer/modules/permissions/bluetooth_permission_descriptor.idl

```
dictionary BluetoothPermissionDescriptor :  PermissionDescriptor {
  DOMString deviceId;
  // These match RequestDeviceOptions. However, for query(), only |deviceId| and
  // |filters| are used.
  sequence<BluetoothLEScanFilterInit> filters;
  sequence<BluetoothServiceUUID> optionalServices = [];
  boolean acceptAllDevices = false;
};
```

The new permission status can be added as the following file:

//third_party/blink/renderer/modules/permissions/bluetooth_permission_status.idl

```
[Exposed=Window]
interface BluetoothPermissionStatus : PermissionStatus {
  attribute FrozenArray<BluetoothDevice> devices;
};
```

The BluetoothPermissionStatus class will implement the PermissionStatus class, and this will require PermissionStatus to not be final. The new class will simply provide a method to access the devices property for JS.

query() API

To add support for Web Bluetooth permissions in the Permissions API, a blink::mojom::BluetoothPermissionDescriptor and blink::mojom::BluetoothPermissionStatus struct needs to be added and the blink::mojom::PermissionStatus enum needs to be refactored into a struct that can be extended with extra data. The following demonstrates the changes to the Web Bluetooth and Permissions API mojom files necessary for the integration:

//third_party/blink/public/platform/modules/bluetooth/web_bluetooth.mojom

```
struct BluetoothPermissionDescriptor {
  string deviceId;
  array<WebBluetoothLeScanFilter>? filters;
  // These two fields are only used when request permissions, so they won't be
  // implemented unless navigator.permissions.request() is launched.
  array<bluetooth.mojom.UUID> optional_services
  bool accept_all_devices;
};

struct BluetoothPermissionStatus {
  array<WebBluetoothDevice> devices;
};
```

//third_party/blink/public/platform/modules/permissions/permission.mojom

```
import "third_party/blink/public/mojom/bluetooth/web_bluetooth.mojom"

enum PermissionName {
  GEOLOCATION,
  // ...
  BLUETOOTH,
};

union PermissionDescriptorExtension {
  // BluetoothPermissionDescriptor is defined in web_bluetooth.mojom so that it can
  // be used in the WebBluetoothService to get permission status.
  BluetoothPermissionDescriptor bluetooth;
  ClipboardPermissionDescriptor clipboard;
  MidiPermissionDescriptor midi;
};
```

//third_party/blink/public/platform/modules/permissions/permission_status.mojom

```
import "third_party/blink/public/mojom/bluetooth/web_bluetooth.mojom"

enum PermissionState {
  GRANTED,
  DENIED,
  ASK,
  LAST = ASK
};

// Unions of possible extensions to the base PermissionResult type.
union PermissionStatusExtension {
  // BluetoothPermissionStatus is defined in web_bluetooth.mojom since the
  // WebBluetoothService produces it.
  BluetoothPermissionStatus bluetooth;
};

struct PermissionStatus {
  PermissionState state;
  PermissionStatusExtension? extension;
};
```

The `query()` method accepts a `PermissionDescriptor` parameter type. For Web Bluetooth, this would be the `BluetoothPermissionDescriptor`. `navigator.permissions.query()` is handled in C++ by `blink::Permissions::query()`, which starts by converting the given permission descriptor parameter into the appropriate type using `ParsePermission()` to check the name of the descriptor and call the appropriate method in `permission_utils.h`. This method will need a check for a descriptor with the name "bluetooth" and create a utility method to create the `BluetoothPermissionDescriptor`.

Once the permission descriptor is parsed, it is sent to the `PermissionService` Mojo interface via the `PermissionService::HasPermission()` method. This interface is implemented by the `PermissionServiceImpl` in the content layer. The descriptor is used in `PermissionServiceImpl::GetPermissionStatus()` to get a `PermissionType` enum value to use for `PermissionServiceImpl::GetPermissionStatusFromType()`. It is here where the code path for Bluetooth permission diverges from the other permission types because the Bluetooth permission descriptor contains extra data that is used for processing the permission status. Therefore, in `GetPermissionStatus()` if the permission type is `PermissionType::BLUETOOTH`, the descriptor needs to be sent to the `WebBluetoothService`. Any other permission types that require special logic to produce a permission status can branch off from this point. The special logic for Web Bluetooth will be handled by `WebBluetoothService::GetPermissionStatus()` described below:

//third_party/blink/public/platform/modules/bluetooth/web_bluetooth.mojom

```
interface WebBluetoothService {
  GetPermissionStatus(BluetoothPermissionDescriptor descriptor)
    => (BluetoothPermissionStatus status);
};
```

This new method will produce a `BluetoothPermissionStatus` that contains the permission state and allowed devices for Web Bluetooth in the current origin. The devices returned are all of the previously paired or currently paired Bluetooth devices, filtered out by the options contained within `BluetoothPermissionDescriptor`.

The descriptor will be sent to `WebBluetoothServiceImpl::GetPermissionStatus()`, which will produce the `BluetoothPermissionStatus`. The `PermissionState` will be determined by the result of `WebBluetoothServiceImpl::GetBluetoothAllowed()`. If the `PermissionState` is denied, the permission status will be produced with an empty `devices` array, even if the current origin does have granted Bluetooth device permissions. However, if the `PermissionState` is granted then the list of allowed devices will be generated by getting the list of all allowed devices with `WebBluetoothServiceImpl::GetDevices()` and then applying the filters to that list. This list will then be returned in the callback.

The `PermissionStatus` is received by `TaskComplete()` in the blink layer. Using the original permission descriptor, if the `PermissionStatus` corresponds to Bluetooth, a `BluetoothPermissionStatus` will be created to resolve the promise with it.

Detecting Previously Connected Devices

The `BluetoothAdapter` class contains a map of devices are paired with, connected to, or have been discovered. This map can be used to retrieve a device, however this map is periodically cleared when a device has not been seen by the adapter for over three minutes. This prevents sites from being able to connect to devices that have been disconnected for over three minutes. Therefore, a new API needs to be implemented that will allow devices to be retrieved by their MAC address from the adapter using platform specific APIs. This API is described by the Device Removal Proposal design document, which provides an idea for a `BluetoothAdapter::RetrievePeripheralFromAddress()` API. Once this API is implemented, it will be possible for sites to reconnect to devices, regardless of how long it has been since they were last connected to.

BluetoothDevice watchAdvertisements() and unwatchAdvertisements()

This API can allow a site to watch for advertisement packets from a device that it has permission to access. This would allow a site to only detect if the specific device comes into range of the Bluetooth adapter without starting a scan for all Bluetooth devices in range. The `unwatchAdvertisements()` and `watchingAdvertisements` API will also be implemented.

The Web Bluetooth spec defines the steps that must be performed when this API is called:
The `watchAdvertisements()` method, when invoked MUST return a new promise `promise` and run the following steps in parallel:

1. Ensure that the UA is scanning for this device's advertisements. The UA SHOULD NOT filter out "duplicate" advertisements for the same device.
2. If the UA fails to enable scanning, reject `promise` with one of the following errors, and abort these steps:
    a. The UA doesn't support scanning for advertisements: `NotSupportedError`
    b. Bluetooth is turned off: `InvalidStateError`
    c. Other reasons: `UnknownError`
3. Queue a task to perform the following steps:
    a. Set `this.watchAdvertisements` to `true`.
    b. Resolve `promise` with `undefined`.

For step one, the `WebBluetoothServiceImpl` needs to start a scan with a filter that only includes the Bluetooth device in which `watchAdvertisements()` was called. This will require a new interface to `WebBluetoothService` which will behave similar to `requestLEScan()` with `keepRepeatedDevices` set to true and the `deviceId` field populated. If an error is encountered while attempting to start the scan, resolve the promise with the appropriate error. Otherwise, add the device to a list of watched devices in the `WebBluetoothServiceImpl` so that the list can be checked when `DeviceAdvertisementReceived()` is called on the service. Next, set the `watchAdvertisements` field of the Bluetooth device to true and resolve the promise with `undefined`.

When the Bluetooth adapter receives an advertisement packet and notifies all of its observers, the `WebBluetoothServiceImpl` should iterate over the list of watched devices and compare the advertisement packet with each device. If a match is found, the Blink layer should be notified to fire an `advertisementreceived` event at the Bluetooth device object that corresponds to the device.

To stop watching for advertisements, the `unwatchAdvertisements()` method can be called. The Web Bluetooth spec defines the steps for this API as follows:
The `unwatchAdvertisements()` method, when invoked, MUST run the following steps:

1. Set `this.watchingAdvertisements` to `false`.

2. If no more BluetoothDevices in the whole UA have watchingAdvertisements set to true, the UA SHOULD stop scanning for advertisements. Otherwise, if no more BluetoothDevices representing the same device as this have watchingAdvertisements set to true, the UA SHOULD reconfigure the scan to avoid receiving reports for this device.

The first step for this API is to set watchingAdvertisements to false. Next, the WebBluetoothServiceImpl should remove the device from the watched devices list and stop the discovery session that corresponds to the device.

Web Bluetooth Mojo Interface Implementation

To implement these two interfaces, web_bluetooth.mojom will include the following changes:

//third_party/blink/public/platform/modules/bluetooth/web_bluetooth.mojom

```
// Rename WebBluetoothScanResult to WebBluetoothAdvertisingEvent to better
// fit its use.

// Remove RequestScanningStartResult, since RequestScanningStart will not
// use it anymore.

interface WebBluetoothService {
  WatchAdvertisementsForDevice(
      WebBluetoothDeviceId device_id,
      pending_associated_remote<WebBluetoothDeviceAdvertisementClient>
          client) => (
          WebBluetoothResult result);
  // Refactor RequestScanningStart
  RequestScanningStart(
      pending_associated_remote<WebBluetoothDeviceAdvertisementClient>
          client,
      WebBluetoothRequestLEScanOptions options) => (
      WebBluetoothResult result);
  UnwatchAdvertisementsForDevice(WebBluetoothDeviceId device_id);
};

// Refactor WebBluetoothScanClient to this to generalize its use.
interface WebBluetoothDeviceAdvertisementClient {
  AdvertisingEvent(WebBluetoothAdvertisingEvent advertisement);
};
```

WebBluetoothService::WatchAdvertisementsForDevice() will accept two parameters, a WebBluetoothDeviceId corresponding to the Bluetooth device to watch and a

WebBluetoothDeviceAdvertisementClient interface so that WebBluetoothService is able to notify the client of matching advertisements.

WebBluetoothService::UnwatchAdvertisementsForDevice() will accept a WebBluetoothDeviceId parameter to cancel the watch for advertisement for that device.

Lastly, WebBluetoothDeviceAdvertisementClient::AdvertisingEvent() will accept a WebBluetoothScanResult structure that contains the advertisement data received for the watched device.

WebBluetoothServiceImpl Implementation

The WebBluetoothService interface is implemented by WebBluetoothServiceImpl. Therefore, the following changes need to be made.

//content/browser/bluetooth/web_bluetooth_service_impl.h

```
class CONTENT_EXPORT WebBluetoothServiceImpl
    : public blink::mojom::WebBluetoothService,
      public WebContentsObserver,
      public BluetoothAdapter::Observer {
 private:
  // The watchAdvertisements() feature can share some of the functionality of
  // this class, so DeviceAdvertisementClient can become the base class for
  // both WatchAdvertisementsClient and ScanningClient.
  class DeviceAdvertisementClient {
   public:
    // Unlike ScanningClient, this will always send the advertising event,
    // since DeviceAdvertisementClient will be destroyed by
    // UnwatchAdvertisementsForDevice().
    virtual bool SendEvent(WebBluetoothAdvertisingEventPtr event);

    // Same as current implementation of
    // ScanningClient::RunRequestScanningStartCallback().
    void RunCallback(WebBluetoothResult result);

   protected:
    DeviceAdvertisementClient(
        mojo::AssociatedRemote<WebBluetoothDeviceAdvertisementClient> client,
        base::OnceCallback<void(WebBluetoothResultPtr)> callback);


   private:
    // Same as ScanningClient implementation.
    void DisconnectionHandler();

    bool disconnected_ = false;
    mojo::AssociatedRemote<WebBluetoothDeviceAdvertisementClient> client_;
    base:OnceCallback<void(WebBluetoothResult)> callback_;
  };

  class WatchAdvertisementsClient : public DeviceAdvertisementClient {
   public:
    WatchAdvertisementsClient(
        mojo::AssociatedRemote<WebBluetoothDeviceAdvertisementClient> client,
        WebBluetoothDeviceId device_id,
        base::OnceCallback<void(WebBluetoothResultPtr)> callback);
    WebBluetoothDeviceId device_id();

   private:
    WebBluetoothDeviceId device_id;
  };

  class ScanningClient : public DeviceAdvertisementClient {
   public:
    ScanningClient(
        mojo::AssociatedRemote<WebBluetoothDeviceAdvertisementClient> client,
        WebBluetoothRequestLEScanOptionsPtr options,
        base::OnceCallback<void(WebBluetoothResultPtr)> callback,
        BluetoothDeviceScanningPromptController* prompt_controller);

    // DeviceAdvertisementClient implementation:
    bool SendEvent(WebBluetoothAdvertisingEventPTr event) override;

    // There is only one use of set_prompt_controller, it should be renamed
    // to ClearPromptController() instead, which will set prompt_controller
    // to nullptr.
    void ClearPromptController();
    BluetoothDeviceScanningPromptController* prompt_controller();
    void set_allow_send_event(bool allow_send_event);
    const WebBluetoothRequestLEScanOptions& scan_options();

   private:
    void AddFilteredDeviceToPrompt(
        const string& device_id,
        const base::Optional<string>& device_name);
  bool allow_send_event_ = false;
  WebBluetoothRequestLeScanOptionsPtr options_;
  BluetoothDeviceScanningPromptController* prompt_controller_;
 };

 // WebContentsObserver methods:
 // ...
 void OnWebContentsLostFocus(RenderWidgetHost* render_widget_host);

 // WebBluetoothService methods:
 // ...
 void WatchAdvertisementsForDevice(
   WebBluetoothDeviceId device_id,
   mojo::PendingAssociatedRemote<WebBluetoothDeviceAdvertisementClient>
     client_info,
   WatchAdvertisementsForDeviceCallback callback) override;
 void UnwatchAdverisetmentsForDevice(
   WebBluetoothDeviceId device_id) override;

 void WatchAdvertisementsForDeviceImpl(
   WebBluetoothDeviceId device_id,
   mojo::AssociatedRemote<WebBluetoothDeviceAdvertisementClient> client,
   WatchAdvertisementsForDeviceCallback callback,
   scoped_refptr<BluetoothAdapter> adapter);

 void OnStartDiscoverySessionForWatchAdvertisements(
   mojo::AssociatedRemote<WebBluetoothDeviceAdvertisementClient> client,
   WebBluetoothDeviceId device_id,
   unique_ptr<BluetoothDiscoverySession> session);

 void MaybeStopDiscovery();

 vector<unique_ptr<WatchAdvertisementsClient>>
   watch_advertisements_clients_;
 std::unique_ptr<BluetoothDiscoverySession>
   watch_advertisements_discovery_session_;
};
```

On a call to `OnWebContentsLostFocus()`, these step will be performed at the end of the current implementation:

1. Clear `watch_advertisements_clients_`.

On a call to `WatchAdvertisementsForDevice()`, these steps will be performed:

1. Perform the same steps as `RequestScanningStart()` to convert `client_info` into a `mojo::AssociatedRemote<WebBluetoothDeviceAdvertisementClient> client` and acquire the `BluetoothAdapter` to pass into the actual implementation of this method, `WatchAdvertisementsForDeviceImpl()`.

`WatchAdvertisementsForDeviceImpl()` will perform these steps:

1. Check that `device_id` and `adapter` are valid.
2. Check that Web Bluetooth is allowed to be used.
3. If there is an existing `watch_advertisements_discovery_session_`:
    a. A `WatchAdvertisementsClient` will be created and added to `watch_advertisements_clients_`.
    b. If not, then create a `BluetoothDiscoveryFilter filter` with the device name, address, and known service UUIDs.
    c. Call `StartDiscoverySessionWithFilter()` on `adapter` with `filter` and the success callback being `OnStartDiscoverySessionForWatchAdvertisements()`.

`OnStartDiscoverySessionForWatchAdvertisements()` will move `session` into `watch_advertisements_discovery_session_` and create the `WatchAdvertisementsClient` to add to `watch_advertisements_clients_`.

On a call to `UnwatchAdvertisementsForDevice()`, these steps will be performed:

1. Find the `WatchAdvertisementsClient` matching `device_id` in `watch_advertisements_clients_`.
2. If one is found, remove it from `watch_advertisements_clients_`
3. Call `MaybeStopDiscovery()` to check if discovery can be stopped.

On a call to `MaybeStopDiscovery()`, these steps will be performed after `scanning_clients_` is checked:

1. if `watch_advertisements_clients_` is empty,
    a. Call `Stop()` on `watch_advertisements_discovery_session_`.
    b. Set `watch_advertisements_discovery_session_` to `nullptr`.

The method will be called at the end of `DeviceAdvertisementReceived()` in place of the existing check for is `scanning_clients_` empty.

Modify `DeviceAdvertisementReceived()` to perform these steps after iterating over `scanning_clients_`:

1. Iterate over a `watch_advertisements_client` in `watch_advertisements_clients_`:
    a. Get the `BluetoothDevice device` corresponding to `watch_advertisements_client->device_id()`.
    b. If `device_address` matches `device->GetAddress()`:

i. Create a `WebBluetoothAdvertisingEventPtr event` from the method's parameters.
ii. Call `watch_advertisement_client->SendEvent(event)`.

Blink IDL Implementation

Once the backend is able to watch for and filter out device advertisements for a particular device, this functionality needs to be exposed in Blink.

//third_party/blink/render/modules/bluetooth/bluetooth_device.idl

```
interface BluetoothDevice : EventTarget {
  [CallWith=ScriptState, RaisesException] Promise<void> watchAdvertisements();
  void unwatchAdvertisements();
  readonly attribute boolean watchingAdvertisements;

  attribute EventHandler onadvertisementreceived;
};
```

//third_party/blink/render/modules/bluetooth/bluetooth_device.h

```
class BluetoothDevice final
    : public EventTargetWithInlineData,
      public ActiveScriptWrappable<BluetoothDevice>,
      public ExecutionContextClient,
      public mojom::blink::WebBluetoothDeviceAdvertisementClient {
 public:
  // IDL exposed interface:
  ScriptPromise watchAdvertisements(ScriptState*, ExceptionState&);
  void unwatchAdvertisements();
  bool watchingAdvertisements();

  // ActiveScriptWrappable implementation:
  bool HasPendingActivity();

  // WebBluetoothDeviceAdvertisementClient implementation:
  void AdvertisingEvent(WebBluetoothDeviceAdvertisementEventPtr event) override;

  DEFINE_ATTRIBUTE_EVENT_LISTENER(advertisementreceived, kAdvertisementreceived)

 private:
  void WatchAdvertisementsCallback(ScriptPromiseResolver*,
                                   WebBluetoothResult);

  mojo::AssociatedReceiver<WebBluetoothDeviceAdvertisementClient>
      client_receiver_;
};
```

On a call to watchAdvertisements(), these steps will be performed:

1. Check that the current context is valid.
2. Create a ScriptPromiseResolver resolver from script_state.
3. If watching_advertisements_ is true, resolve resolver with undefined and return.
4. Store promise from resolver->Promise().
5. Create a mojo::PendingAssociatedRemote<WebBluetoothDeviceAdvertisementClient> client.
6. Bind client to client_receiver_.
7. Call bluetooth_->Service()->WatchAdvertisementsForDevice() using device_id, client, and a bound callback to WatchAdvertisementsCallback() with this and resolver passed to it.

WatchAdvertisementsCallback() will perform these steps:

1. Verify that context from resolver is still valid.
2. If result is an error, reject resolver.
3. Resolve resolver with undefined.

On a call to unwatchAdvertisements(), these steps will be performed:

1. Check that the current context is valid.
2. If watching_advertisements_ is false, return.
3. Call bluetooth_->Service()->UnwatchAdvertisementsForDevice() using device_id.
4. Reset client_receiver_.

watchingAdvertisements() will return true if client_receiver_ is bound.

On a call to HasPendingActivity(), these steps will be performed:

1. Return true if context is valid and HasEventListeners() is true.

On a call to AdvertisingEvent(), these steps will be performed:

1. Check that the current context is valid.
2. Create a BluetoothDevice bluetooth_device from event.
3. Create a BluetoothAdvertisingEvent advertising_event from bluetooth_device and result.
4. DispatchEvent(*advertising_event).

BluetoothDiscoveryFilter Refactor

For watchAdvertisements(), the most effective filter will be the device address, since device names can be shared. The BluetoothDiscoveryFilter will have the following additions:

//device/bluetooth/bluetooth_discovery_filter.h

```
class DEVICE_BLUETOOTH_EXPORT BluetoothDiscoveryFilter {
 public:
  struct DEVICE_BLUETOOTH_EXPORT DeviceInfoFilter {
    std::string address;
  };
};
```

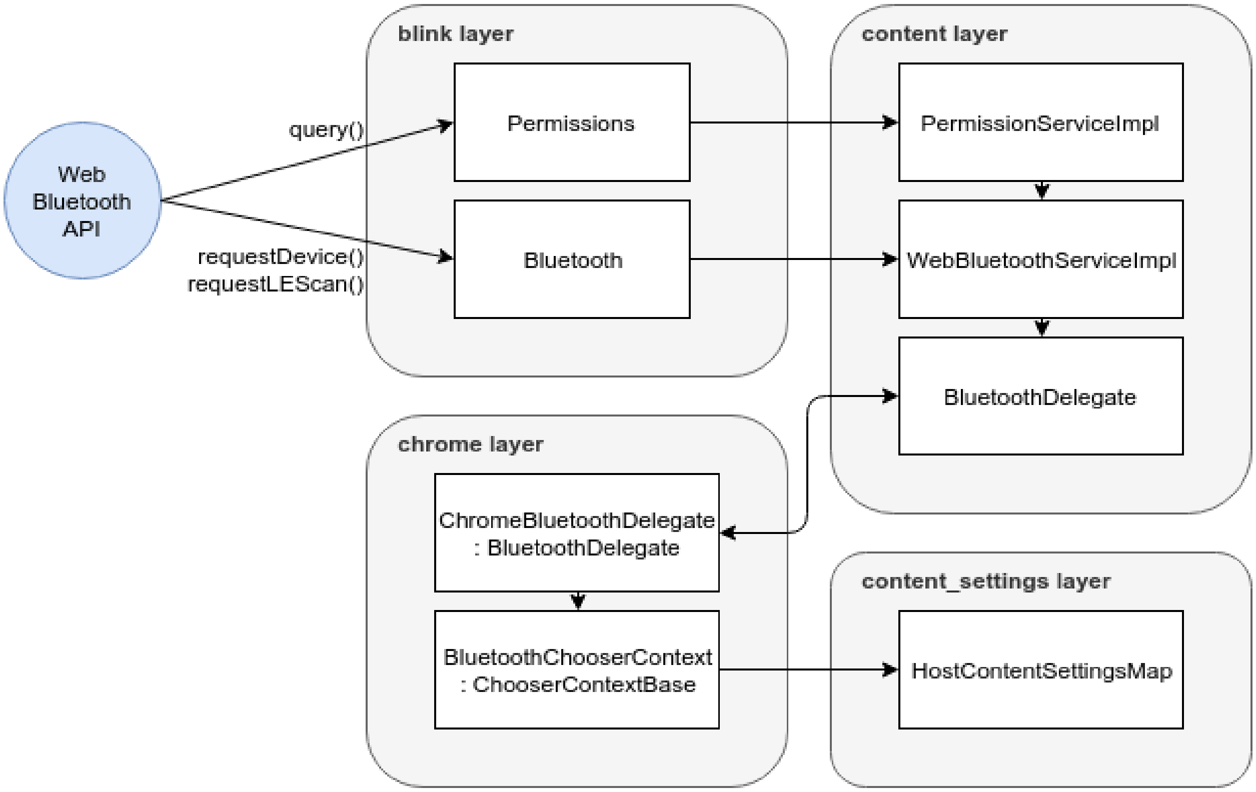
The BluetoothDiscoveryFilter::DeviceInfoFilter class will contain a new address field for the device MAC address. The equality operator methods will be updated to take this new field into account.

Each platform's implementation of Bluetooth discovery will require an update to make use of the device address filter.

- Android
  - android.bluetooth.le.ScanFilter can accept a device address filter.
  - Update `ChromeBluetoothScanFilterBuilder` to use the new `DeviceInfoFilter` field.
- Windows
  - The BluetoothDiscoveryFilter seems to be unused in bluetooth_adapter_winrt.cc
  - A `winrt::Windows::Devices::Bluetooth::Advertisement::BluetoothLEAdvertisement` will need to be created from the properties in BluetoothDiscoveryFilter
  - There is not a specific field for device address, so perhaps they can be set in the raw `DataSections` of the `BluetoothLEAdvertisement` object.
    - `0x1B` is the value for LE Bluetooth Device Address (https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile/), a MAC address is a fixed 6 bytes.
    - Create a `BluetoothLEAdvertisementBytePattern` with its `Data` property set to the MAC address and the `DataType` set to `0x1B`.
    - There seems to be two ways to set a filter for device address.
      - `BluetoothLEAdvertisementFilter::BytePatterns()`.
      - `BluetoothLEAdvertisementFilter::Advertisement()`, with a `BluetoothLEAdvertisement` that contains a `BluetoothLEAdvertisementDataSection` with the correct `Data` and `DataType` for the MAC address.
  - The device name can be set with `BluetoothLEAdvertisement::LocalName()`.
  - The services can be set with `BluetoothLEAdvertisement::ServiceUuids()`.
- Linux
  - `BluetoothAdapterBlueZ::SetDiscoveryFilter()` constructs the `bluez::BluetoothAdapterClient::DiscoveryFilter`. It looks like the device address can be set in the Pattern property.

- macOS
    - CBCentralManager::scanForPeripherals() initiates a scan for Bluetooth devices, however it only provides the ability to filter devices using service UUIDs. The scan would need to be started with the service UUIDs of the device in question, and then perform further filtering when the advertisement is received. This will be done in `WebBluetoothServiceImpl::DeviceAdvertisementReceived()`.

Refactor to use ChooserContextBase



High-level overview of how permissions will work using a `BluetoothChooserContext`.

With persistent device permissions, it is important to allow users to control these permissions. This can be achieved by refactoring the Web Bluetooth permissions systems to store permissions using a `ChooserContextBase`. The refactor has a couple of advantages. The first advantage is that the `ChooserContextBase` class is already supported by Site Settings and Page Info, so minimal changes are required to be able to display Bluetooth device permissions in these UIs. The second advantage is that it will make Bluetooth device permissions homogenous with the other device API permissions. The WebUSB and WebSerial permissions systems will be used as references for implementing the permissions system for Web Bluetooth. The current Web Bluetooth permissions are stored in the content/ directory, but the `ChooserContextBase` class is under the chrome/ directory. To cross the boundary between these two directories, an abstract

BluetoothDelegate class that is a public export of the content layer is needed. Then, a ChromeBluetoothDelegate class can implement BluetoothDelegate to provide a bridge between content/ and chrome/.

BluetoothChooserContext

The BluetoothChooserContext class will inherit from the ChooserContextBase class and implement the following methods:

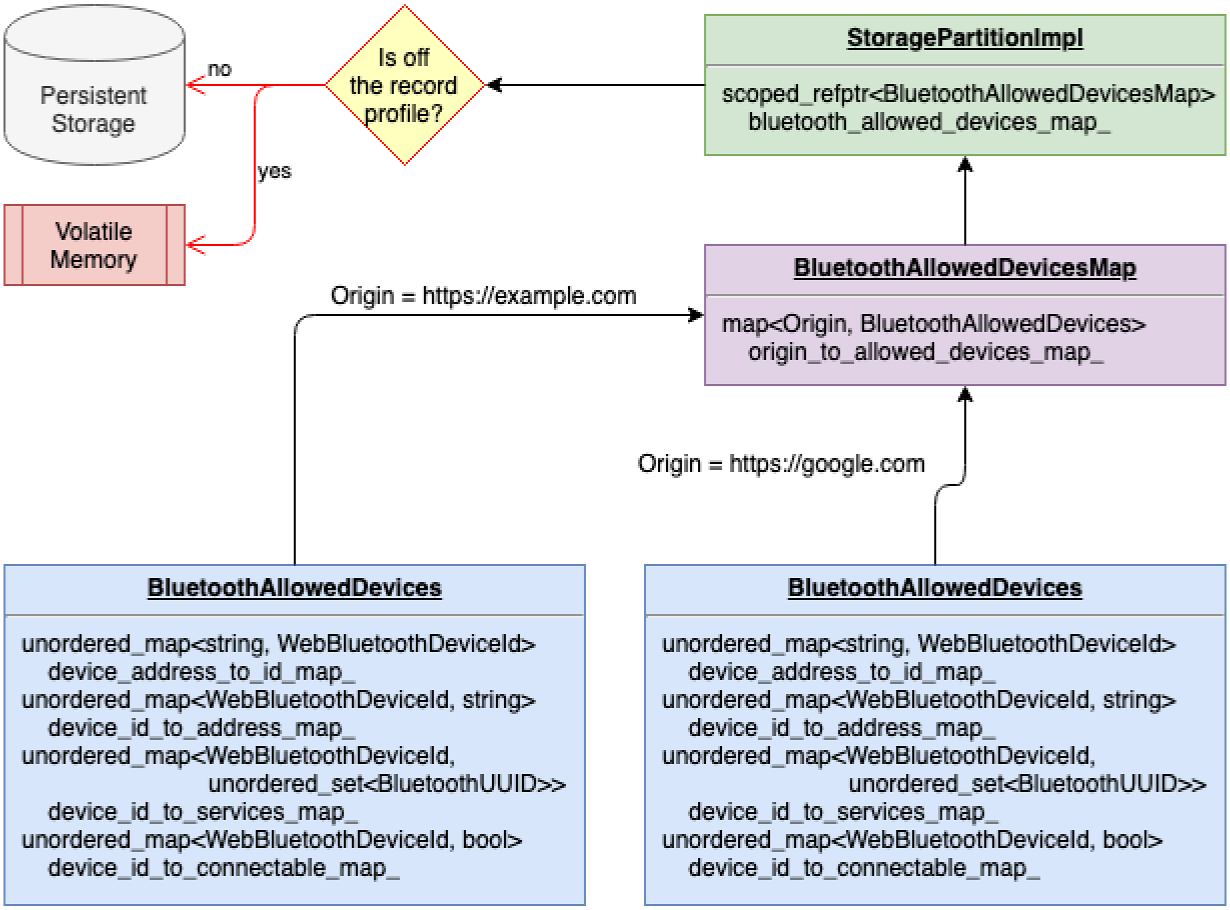
//chrome/browser/bluetooth/bluetooth_chooser_context.h

```
class BluetoothChooserContext : public ChooserContextBase {
 public:
  explicit BluetoothChooserContext(Profile* profile);
  ~BluetoothChooserContext() override;
  WebBluetoothDeviceId GetWebBluetoothDeviceId(
      const Origin& requesting_origin,
      const Origin& embedding_origin,
      const string& device_address);
  string GetDeviceAddress(
      const Origin& requesting_origin,
      const Origin& embedding_origin,
      const WebBluetoothDeviceId& device_id);
  WebBluetoothDeviceId AddScannedDevice(
      const Origin& requesting_origin,
      const Origin& embedding_origin,
      const std::string& device_address);
  WebBluetoothDeviceId GrantServiceAccessPermission(
      const Origin& requesting_origin,
      const Origin& embedding_origin,
      const BluetoothDevice* device,
      const WebBluetoothRequestDeviceOptionsPtr* options);
  bool HasDevicePermission(
      const Origin& requesting_origin,
      const Origin& embedding_origin,
      const WebBluetoothDeviceId& device_id);
  bool IsAllowedToAccessService(
      const Origin& requesting_origin,
      const Origin& embedding_origin,
      const WebBluetoothDeviceId& device_id,
      const BluetoothUUID& service);
  bool IsAllowedToAccessAtLeastOneService(
      const Origin& requesting_origin,
      const Origin& embedding_origin,
      const WebBluetoothDeviceId& device_id);

 private:
  const bool is_incognito_;
};
```

The `BluetoothChooserContext` will provide the interface to grant or query the permissions for a Bluetooth device for a set of requesting and embedding origins. Additionally, the various settings UIs use the `ChooserContextBase::GetGrantedObjects()` and `ChooserContextBase::GetAllGrantedObjects()` methods to retrieve the chooser based permissions for display. Therefore, the addition of this class will facilitate the ability to display what Bluetooth devices are allowed to be accessed by which pair or origins.

Storing Persistent Web Bluetooth Permissions



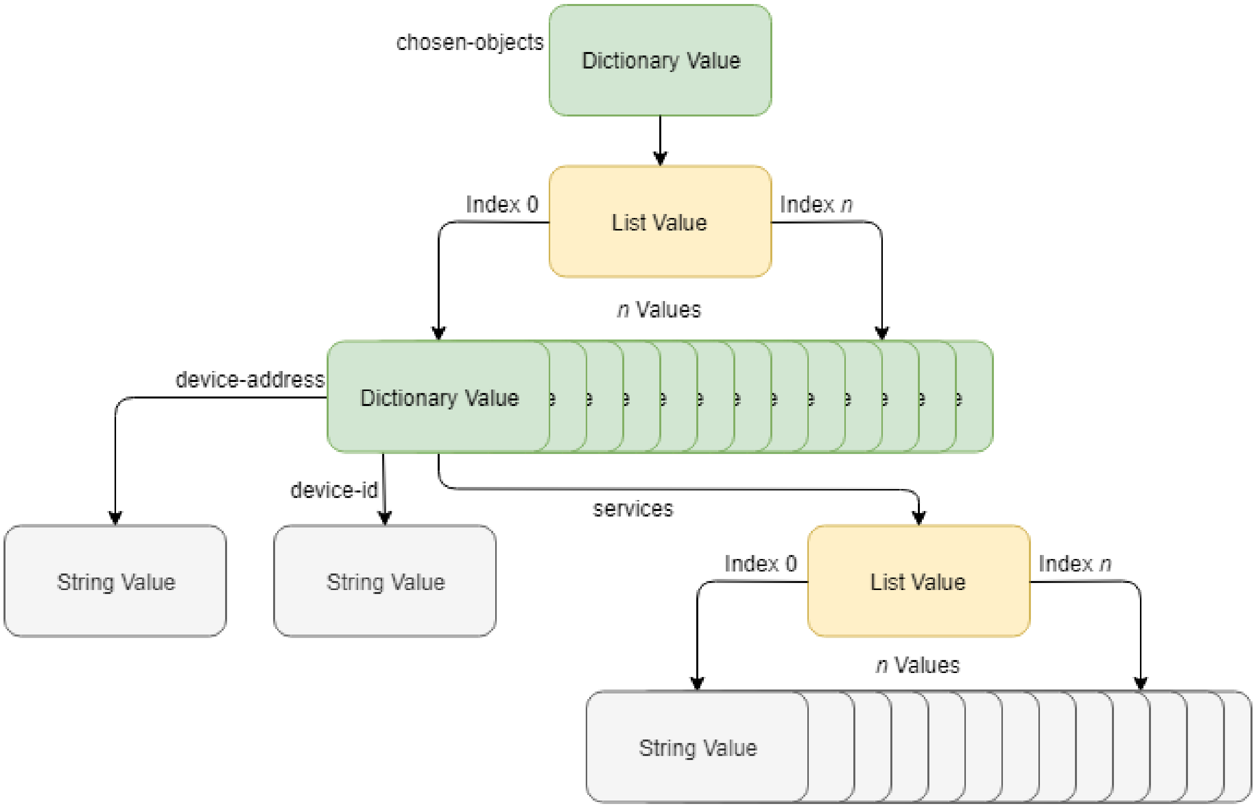
Persistent Storage
no
Is off the record profile?
yes
Volatile Memory
StoragePartitionImpl
scoped_refptr<BluetoothAllowedDevicesMap>
bluetooth_allowed_devices_map_
BluetoothAllowedDevicesMap
map<Origin, BluetoothAllowedDevices>
origin_to_allowed_devices_map_
Origin = https://example.com
Origin = https://google.com
BluetoothAllowedDevices
unordered_map<string, WebBluetoothDeviceId>
device_address_to_id_map_
unordered_map<WebBluetoothDeviceId, string>
device_id_to_address_map_
unordered_map<WebBluetoothDeviceId, unordered_set<BluetoothUUID>>
device_id_to_services_map_
unordered_map<WebBluetoothDeviceId, bool>
device_id_to_connectable_map_
BluetoothAllowedDevices
unordered_map<string, WebBluetoothDeviceId>
device_address_to_id_map_
unordered_map<WebBluetoothDeviceId, string>
device_id_to_address_map_
unordered_map<WebBluetoothDeviceId, unordered_set<BluetoothUUID>>
device_id_to_services_map_
unordered_map<WebBluetoothDeviceId, bool>
device_id_to_connectable_map_

Permissions storage model with BluetoothAllowedDevices.

Web Bluetooth generates a `WebBluetoothDeviceId` for each `WebBluetoothDevice` so that the device can be identified without exposing its MAC address. The current method that it uses to create `WebBluetoothDeviceIds` is to generate a random 128-bit string, and then base64 encode that string. These IDs are stored in the `BluetoothAllowedDevices` class in two separate maps for `WebBluetoothDeviceId` to the device address and vice versa. Each origin has its own

`BluetoothAllowedDevices` in the `BluetoothAllowedDevicesMap`. The `BluetoothAllowedDevicesMap` is stored in a `StoragePartition` that is the default browser context storage partition. The data in the storage partition is persisted if the current browser context is not an off-the-record context.

The `WebBluetoothDeviceIds` are created for a device in two cases. The first is through the `requestDevice()` API after a user selects a Bluetooth device from the chooser prompt. This ID and corresponding device address are stored along with the services provided in the filters for `requestDevice()`, and the device is marked as connectable. The second is through the `requestLEScan()` API after a device advertisement is received. As with the first case, the ID and corresponding device address are stored but no services are stored and the device is not marked as connectable.



This diagram demonstrates the structure of the `base::Value` objects that will store Bluetooth permissions when the ID generation algorithm is completely random.

The `ChooserContextBase` stores permissions into the `HostContentSettingsMap`, which stores preferences for a profile. This allows the permissions to be persistent across browsing sessions on the same machine. To store the permissions, Web Bluetooth will need two new `ContentSettingsType` enums named `CONTENT_SETTINGS_TYPE_BLUETOOTH_GUARD` and `CONTENT_SETTINGS_TYPE_BLUETOOTH_CHOOSER_DATA`. The first content settings type is used to toggle the ability to use the Web Bluetooth API. The second content settings type is used to store Bluetooth device permissions for each site. This setting unique for each pair of requesting and embedding origins and it is stored as a dictionary type `base::Value` object. For each Bluetooth device permission, this `base::Value` object will store the ID, address, and list of allowed services as shown in the diagram above.

`GrantServiceAccessPermission()` stores Bluetooth device permissions granted for a site. The permission is stored into `HostContentSettingsMap` using the base class `GrantObjectPermission()` method where it will be persisted until the user revokes the permission.

Checking Web Bluetooth Permissions

The `HasDevicePermission()` method provides the ability to check Web Bluetooth device permissions. This method gets the object list for the current site and iterates over the list to find a matching `WebBluetoothDeviceId`. If a match is found, that means that the site does have permission to use the device and the method returns true.

Web Bluetooth also requires a site to have permission to use a GATT service, so the `IsAllowedToAccessService()` method provides the ability to check these permissions. This method also iterates through the object list for the current site to find a matching `WebBluetoothDeviceId`. If an object with `WebBluetoothDeviceId` is found, then the services stored in the permission object is iterated over to find a matching `BluetoothUUID`. If a match for the service is found, then the site does have permission to access the service and the method returns true.

There is a special error returned by the Web Bluetooth API when a site has permission to use a Bluetooth device, but not any of its services. The `IsAllowedToAccessAtLeastOneService()` method iterates over the object list to find an object that contains a matching `WebBluetoothDeviceId`. If a match is found, then the services list of the object is checked to see if it's not empty and this result is returned.

There is one case where a device address needs to be retrieved from a device ID in the Web Bluetooth service. The `GetDeviceAddress()` method provides this capability by iterating over the object list to find an object with a matching `WebBluetoothDeviceId`. If a match is found, the corresponding address is returned. Otherwise, an empty string is returned.

For the Web Bluetooth Scanning API, the `AddScannedDevice()` method can be used to generate a `WebBluetoothDeviceId` without granting permission to the Web Bluetooth API to GATT connect to the device. If the device is granted permission through `navigator.bluetooth.requestDevice()`, then the ID that was generated with `AddScannedDevice()` is stored in persistent storage. `AddScannedDevice()` will then return this persistent ID the next time that the device is detected.

BluetoothDelegate

The BluetoothDelegate provides an interface between the chrome and content layers by being an abstract class in the content layer that can be implemented in the chrome layer. The BluetoothDelegate class has the following interface:

//content/public/browser/bluetooth_delegate.h

```
class CONTENT_EXPORT BluetoothDelegate {
 public:
  struct PermittedDevice {
    WebBluetoothDeviceId device_id;
    string device_name;
  };
  virtual ~BluetoothDelegate() = default;
  virtual WebBluetoothDeviceId GetWebBluetoothDeviceId(
      RenderFrameHost* frame,
      string device_address) = 0;
  virtual string GetDeviceAddress(
      RenderFrameHost* frame,
      WebBluetoothDeviceId device_id) = 0;
  virtual WebBluetoothDeviceId AddScannedDevice(
      RenderFrameHost* frame,
      const std::string& device_address) = 0;
  virtual WebBluetoothDeviceId GrantServiceAccessPermission(
      RenderFrameHost* frame,
      const BluetoothDevice* device,
      const WebBluetoothRequestDeviceOptionsPtr& options) = 0;
  virtual bool HasDevicePermission(
      RenderFrameHost* frame,
      WebBluetoothDeviceId device_id) = 0;
  virtual bool IsAllowedToAccessService(
      RenderFrameHost* frame,
      WebBluetoothDeviceId device_id,
      BluetoothUUID service) = 0;
  virtual bool IsAllowedToAccessAtLeastOneService(
      RenderFrameHost* frame,
      WebBluetoothDeviceId device_id) = 0;
  virtual vector<PermittedDevice> GetPermittedDevices(RenderFrameHost* frame)
      = 0;
};
```

ChromeBluetoothDelegate

The ChromeBluetoothDelegate implements BluetoothDelegate, and it lives in //chrome/browser/bluetooth/. The class allows the WebBluetoothServiceImpl and BluetoothChooserContext to interact with each other.

The methods grab the requesting and embedding origins from the `RenderFrameHost*` to pass to the methods of the same name in `BluetoothChooserContext`. The `GetDevices()` method instead calls `BluetoothChooserContext::GetGrantedObjects()` to create `WebBluetoothDevice` objects from the `base::Value` objects returned by that method.

Deprecating BluetoothAllowedDevices

Once the `BluetoothChooserContext` and `BluetoothDelegate` classes are implemented, they can replace permissions checks in `WebBluetoothServiceImpl`, which are handled by `BluetoothAllowedDevices`.

WebBluetoothServiceImpl::IsDevicePaired()

This method checks if `BluetoothAllowedDevices` contains a mapping of the given device address to a device ID. To make this method work with `BluetoothChooserContext`, it simply needs to call `BluetoothDelegate::HasDevicePermission()` with the device address.

WebBluetoothServiceImpl::DeviceAdvertisementReceived()

This method is overridden from `BluetoothAdapter::Observer`, and it is called when the Bluetooth adapter receives advertisement packets from nearby Bluetooth devices. When there are active `ScanningClients`, detected devices are added to `BluetoothAllowedDevices`. Using the new permissions storage model, this logic can be replaced with a call to `BluetoothDelegate::AddScannedDevice()` to return in the `blink::mojom::WebBluetoothScanResult`.

WebBluetoothServiceImpl::RemoteServerConnect()

`BluetoothAllowedDevices` is used in this method to check if it is allowed to make a GATT connection to a device with the given ID. This check was added after devices detected by the Web Bluetooth Scanning API were added to the maps in `BluetoothAllowedDevices` so that they had a `WebBluetoothDeviceId`. Prior to that change, a `WebBluetoothDeviceId` was guaranteed to correspond to a Bluetooth device that was paired. With `BluetoothChooserContext`, this check can be done with a call to `BluetoothDelegate::HasDevicePermission()` using the device ID.

WebBluetoothServiceImpl::RemoteServerGetPrimaryServices()

This method uses `BluetoothAllowedDevices` twice. The first use is to check if the device with `device_id` is allowed to access at least one service. The second use is to check if the device with `device_id` is allowed to access the service passed into this method. The two checks are necessary because different errors are returned if the checks fail. Therefore, the first check can be done with a call to `BluetoothDelegate::IsAllowedToAccessAtLeastOneService()`, while the second check can be done with `BluetoothDelegate::HasDevicePermission()`.

WebBluetoothServiceImpl::RemoteServerGetPrimaryServicesImpl()

This method simply uses `BluetoothAllowedDevices` to check if each of the primary GATT services is able to be accessed, and ignore the services that are not allowed. Therefore, this can be done with `BluetoothDelegate::HasDevicePermission()` check for each primary GATT service.

WebBluetoothServiceImpl::OnGetDeviceSuccess()

This method is the success callback that is run after the user selects a Bluetooth device from the chooser prompt. `BluetoothAllowedDevices` is used here to add the selected device's address and requested services to the maps, essentially granting permission to use them. This logic can be replaced with `BluetoothDelegate::GrantServiceAccessPermission()` with the device address, the services requested, and whether the user chose to grant the permission temporarily or not.

WebBluetoothServiceImpl::QueryCacheForDevice()

This method tries to find the `BluetoothDevice` object that corresponds to the given `WebBluetoothDeviceId` from the `BluetoothAdapter`. Before it is able to query the adapter for the device, the ID needs to be used to get the address of the device. This is simply replaced with a call to `BluetoothDelegate::GetDeviceAddress()`.

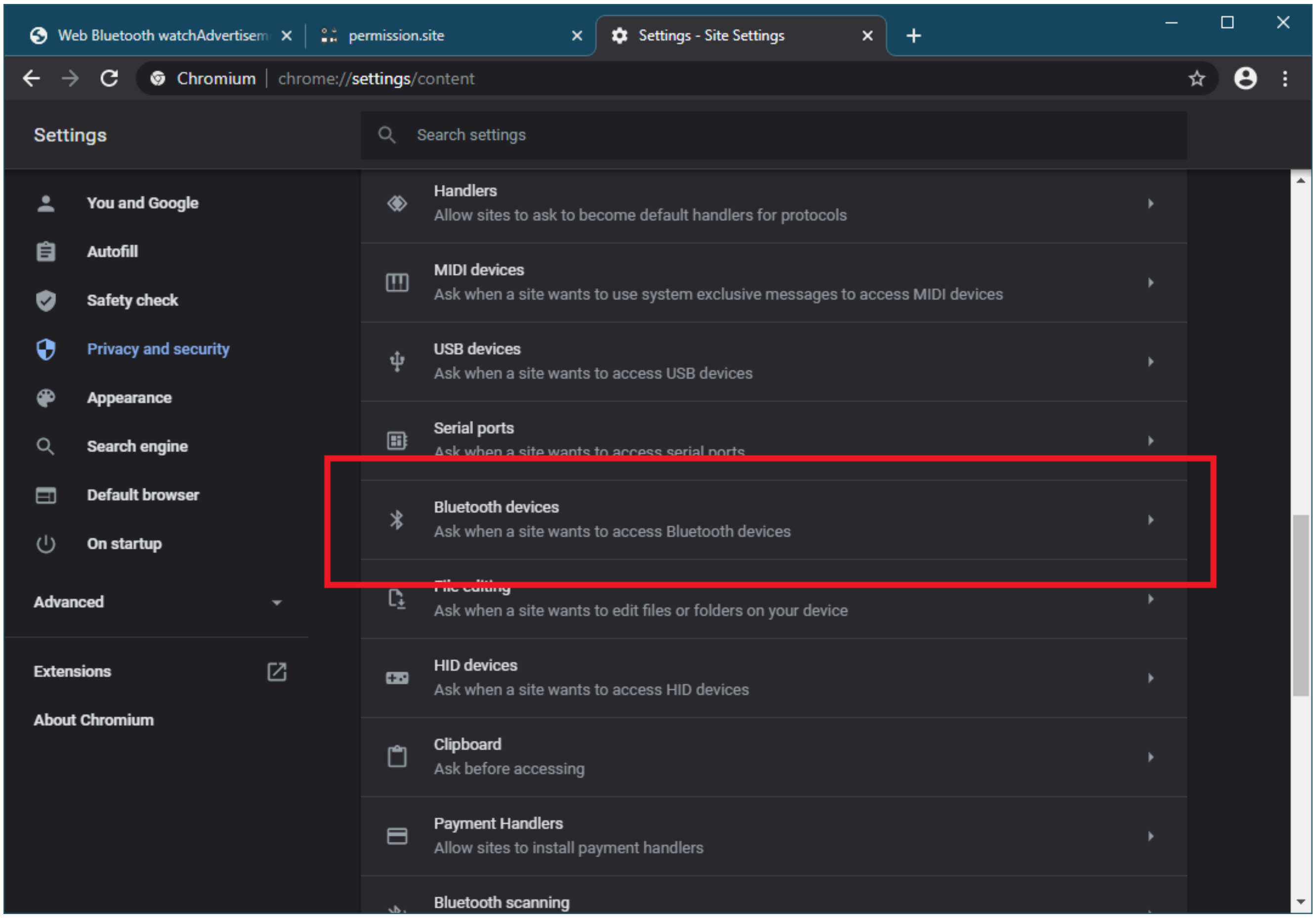
WebBluetoothServiceImpl::GetDevices()

The list of devices can be grabbed using `BluetoothDelegate::GetPermittedDevices()`. The method will return a vector \`PermittedDevice\` structs which contain the device ID and name in order to contruct the WebBluetoothDevice objects needed by the corresponding JavaScript API.

UI Changes

Desktop Site Settings

Once Web Bluetooth is using `ChooserContextBase`, getting the Bluetooth permissions to show up in Site Settings and Page Info is trivial. For the desktop Site Settings WebUI page, the following changes need to be done. First, the appropriate `ContentSettingsTypeNameEntry` and `ChooserTypeNameEntry` entries need to be added to `kContentSettingsTypeGroupNames[]` and `kChooserTypeGroupNames[]` respectively in site_settings_helper.cc. These changes also need to be reflected on the WebUI side in constants.js. Then a new entry needs to be added to the chrome://settings/content page by modifying the privacy_page.html to add an entry similar to the one that already exists for WebUSB. The chooser_exception_list.js file should add a case to `chooserTypeChanged_()` to display the appropriate empty list message for Bluetooth devices.



Handlers
Allow sites to ask to become default handlers for protocols

MIDI devices
Ask when a site wants to use system exclusive messages to access MIDI devices

USB devices
Ask when a site wants to access USB devices

Serial ports
Ask when a site wants to access serial ports

Bluetooth devices
Ask when a site wants to access Bluetooth devices

File editing
Ask when a site wants to edit files or folders on your device

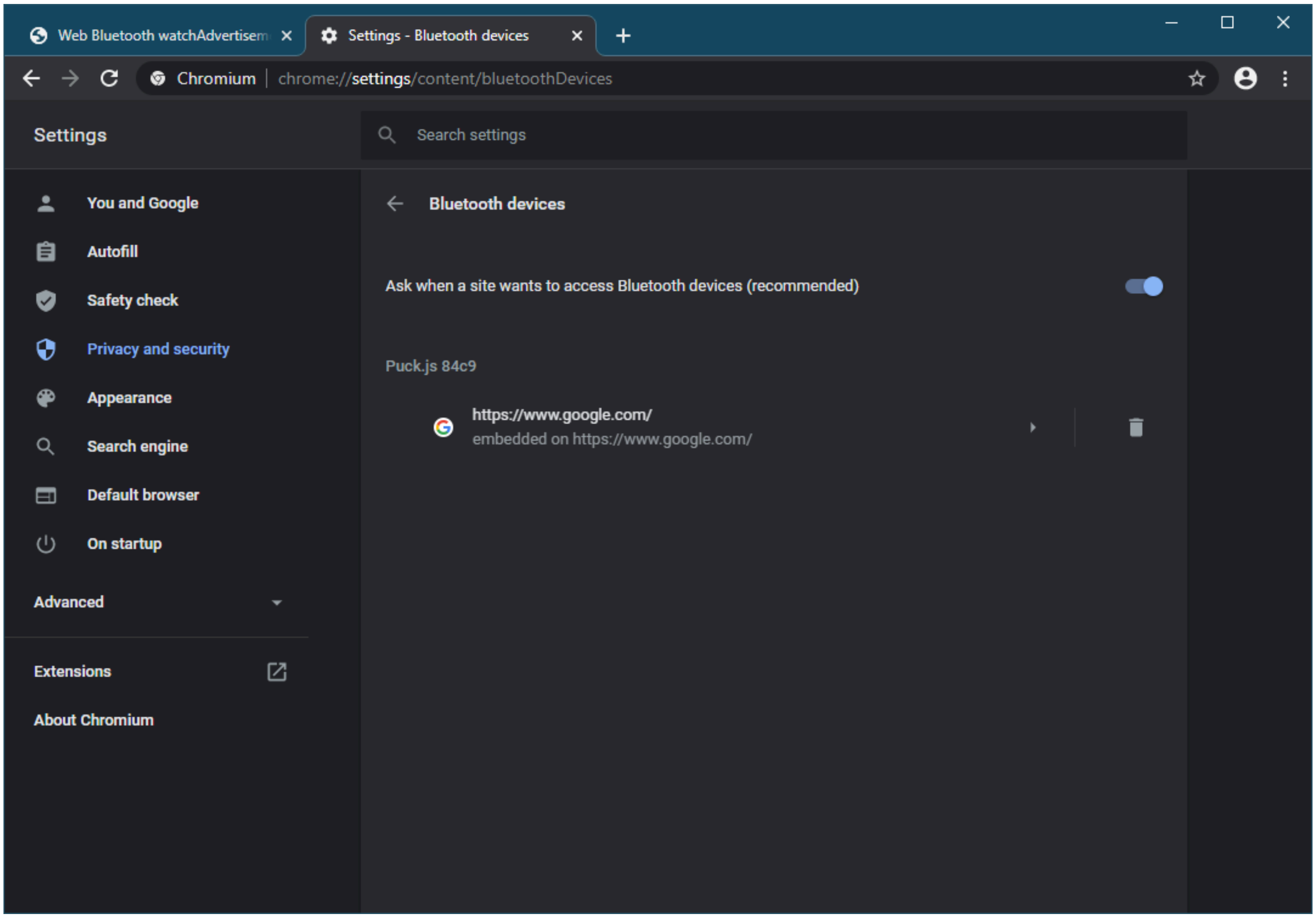
HID devices
Ask when a site wants to access HID devices

Clipboard
Ask before accessing

Payment Handlers
Allow sites to install payment handlers

Bluetooth scanning

Bluetooth devices entry in Site Settings page.



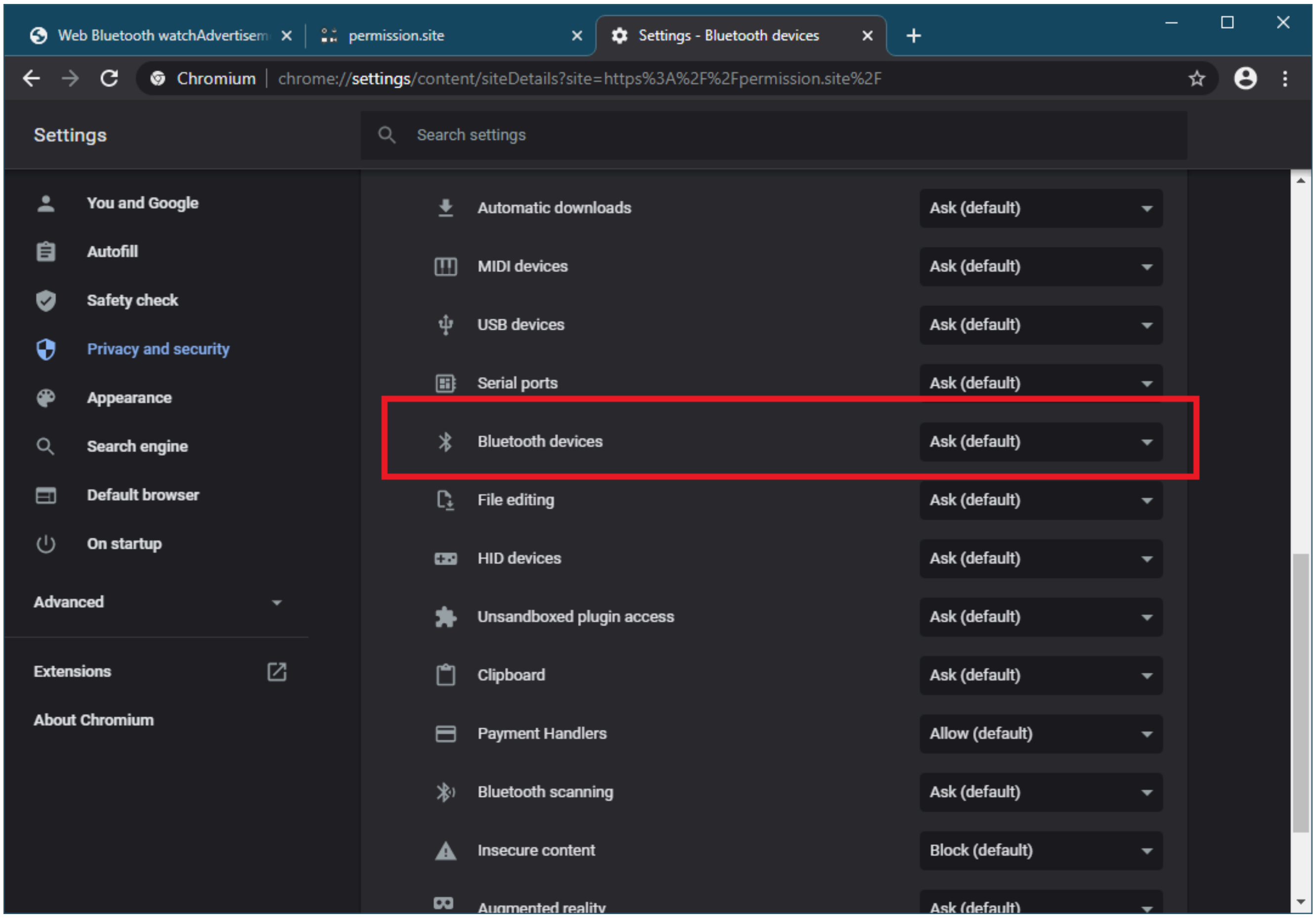
Bluetooth devices

Ask when a site wants to access Bluetooth devices (recommended) ●

Puck.js 84c9

https://www.google.com/
embedded on https://www.google.com/

Bluetooth device permission page in Site Settings. Users will be able to revoke site access to devices here or block Web Bluetooth entirely.



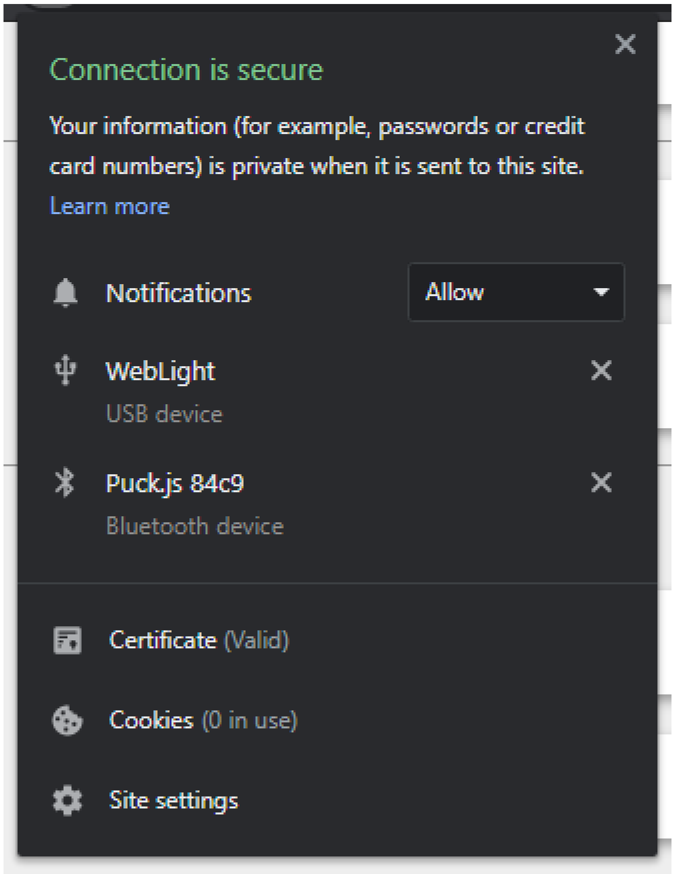
chrome://settings/content/siteDetails?site=https%3A%2F%2Fpermission.site%2F

Automatic downloads	Ask (default)
MIDI devices	Ask (default)
USB devices	Ask (default)
Serial ports	Ask (default)
Bluetooth devices	Ask (default)
File editing	Ask (default)
HID devices	Ask (default)
Unsandboxed plugin access	Ask (default)
Clipboard	Ask (default)
Payment Handlers	Allow (default)
Bluetooth scanning	Ask (default)
Insecure content	Block (default)
Augmented reality	Ask (default)

Bluetooth devices entry in Site Details page. Users will be able to block Web Bluetooth for a given site in this menu.

Desktop Page Info

The Page Info dialog box is also fairly trivial to implement. The only change needed is to add an appropriate `ChooserUIInfo` entry needs to be created for `kChooserUIInfo[]`.



Connection is secure

Your information (for example, passwords or credit card numbers) is private when it is sent to this site.
Learn more

Notifications Allow

WebLight
USB device

Puck.js 84c9
Bluetooth device

Certificate (Valid)

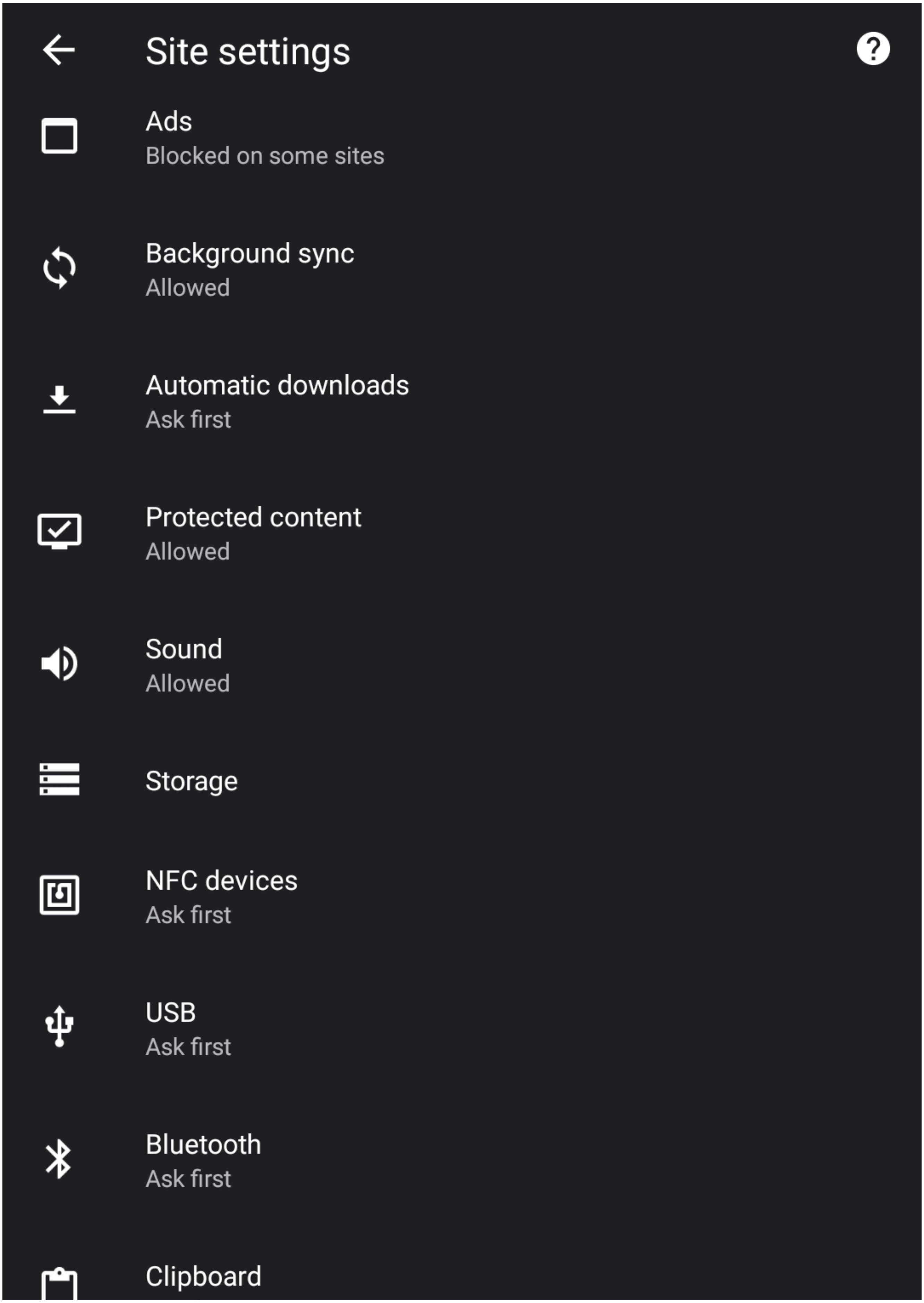
Cookies (0 in use)

Site settings

Bluetooth device permissions in Page Info. Users will be able to quickly revoke device permissions for the current site or block Web Bluetooth for the site.

Android Site Settings

The Android Site Settings is again fairly trivial to implement. The `SiteSettingsCategory` class will need to be updated with support for the `CONTENT_SETTINGS_TYPE_BLUETOOTH_GUARD` and `CONTENT_SETTINGS_TYPE_BLUETOOTH_CHOOSER_DATA` content settings types. The `SiteSettingsPreferences` class implements the `PreferenceFragment` that renders all of the site settings, therefore it needs to be updated to display Bluetooth settings as well. The `ContentSettingsResources` class is used to retrieve the assets used by a specific setting, such as the USB icon for `CONTENT_SETTINGS_TYPE_USB_GUARD`, therefore this class also needs to be updated to be able to return Bluetooth settings assets.



Site settings

Ads
Blocked on some sites

Background sync
Allowed

Automatic downloads
Ask first

Protected content
Allowed

Sound
Allowed

Storage

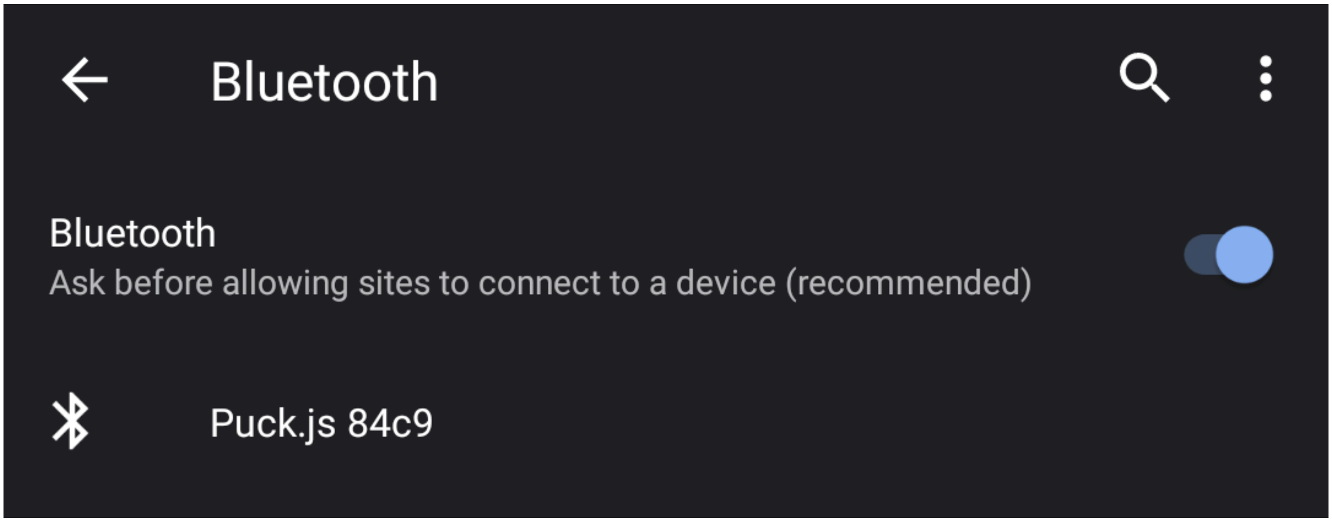
NFC devices
Ask first

USB
Ask first

Bluetooth
Ask first

Clipboard

Bluetooth entry in the Site Settings menu.

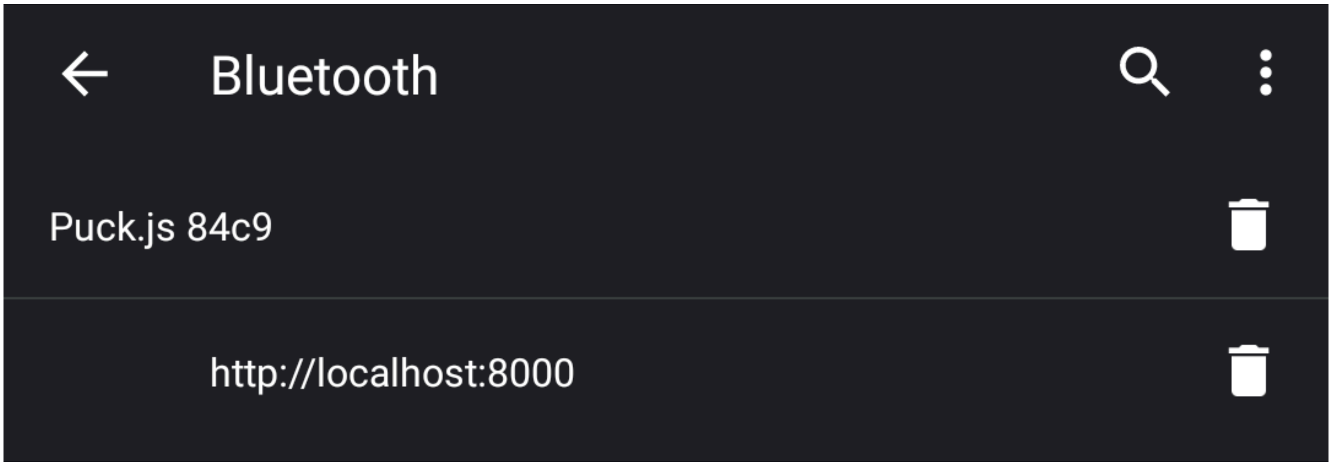


Bluetooth

Bluetooth
Ask before allowing sites to connect to a device (recommended)

Puck.js 84c9

First Bluetooth permissions screen that is displayed after tapping on the Bluetooth entry in Site Settings. This screen groups Bluetooth device permissions under the device's name.

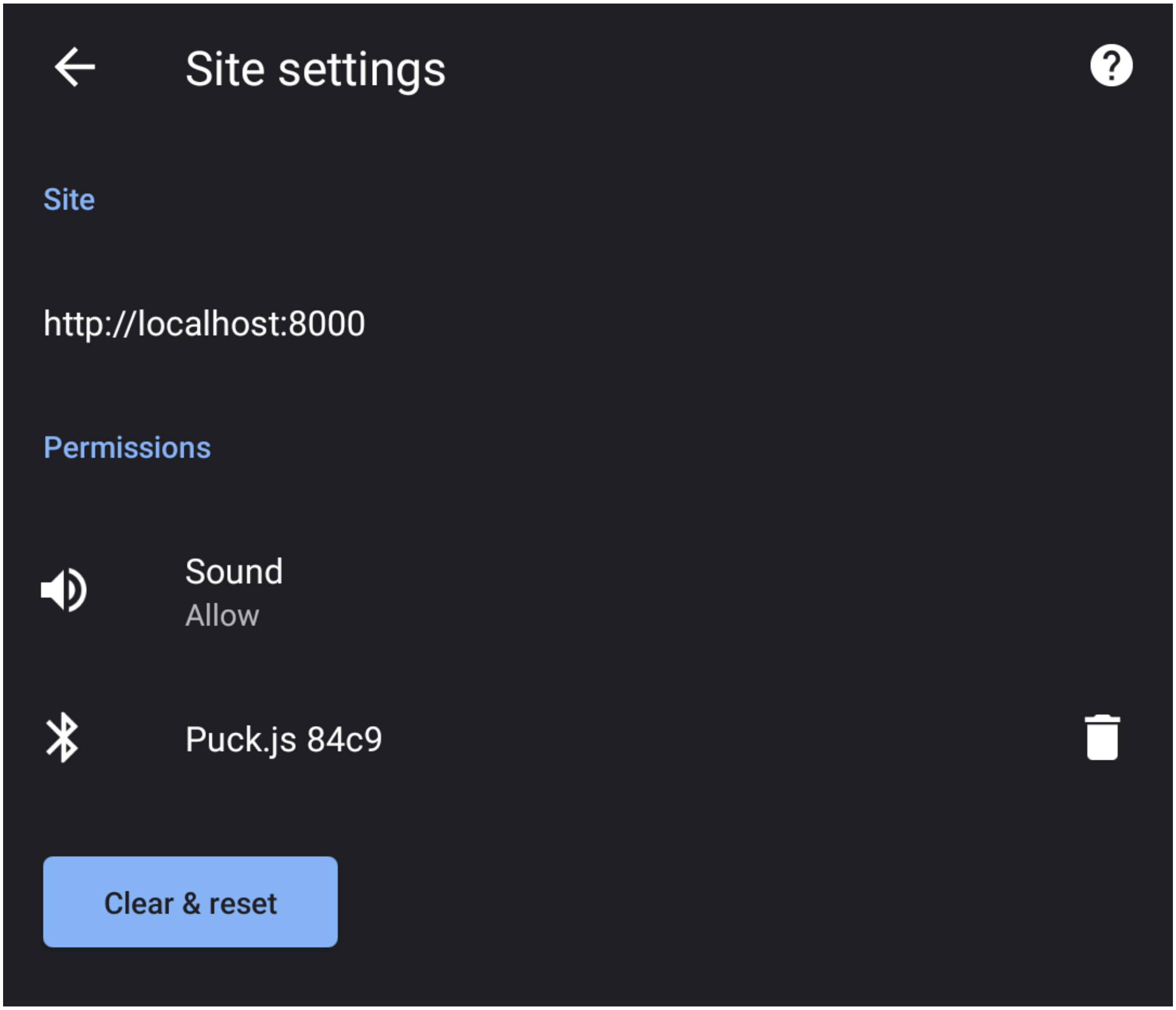


Bluetooth

Puck.js 84c9

http://localhost:8000

Second Bluetooth permissions screen that is displayed after tapping on a Bluetooth device name. This screen displays all of the sites that are allowed to connect to the device.



Site settings

Site

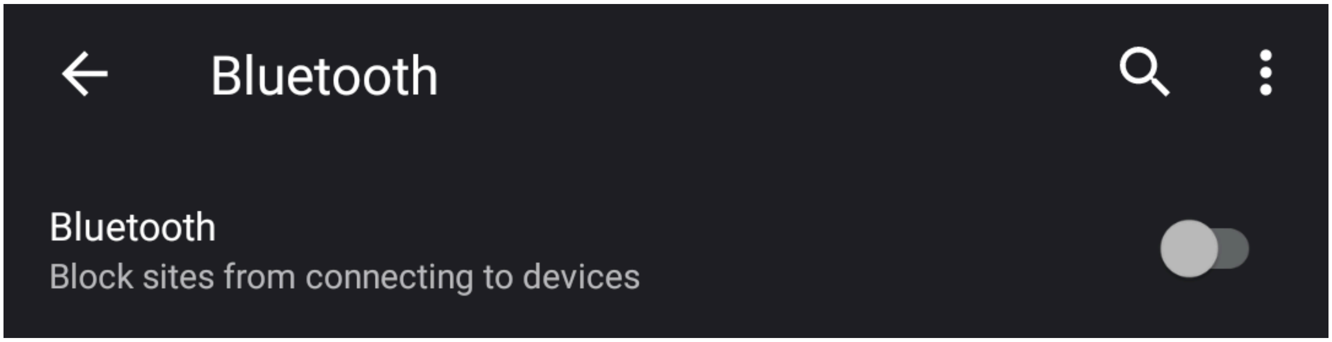
http://localhost:8000

Permissions

Sound
Allow

Puck.js 84c9

Site details screen that is displayed after tapping on a site URL. This screen displays all of the permissions that the site has.



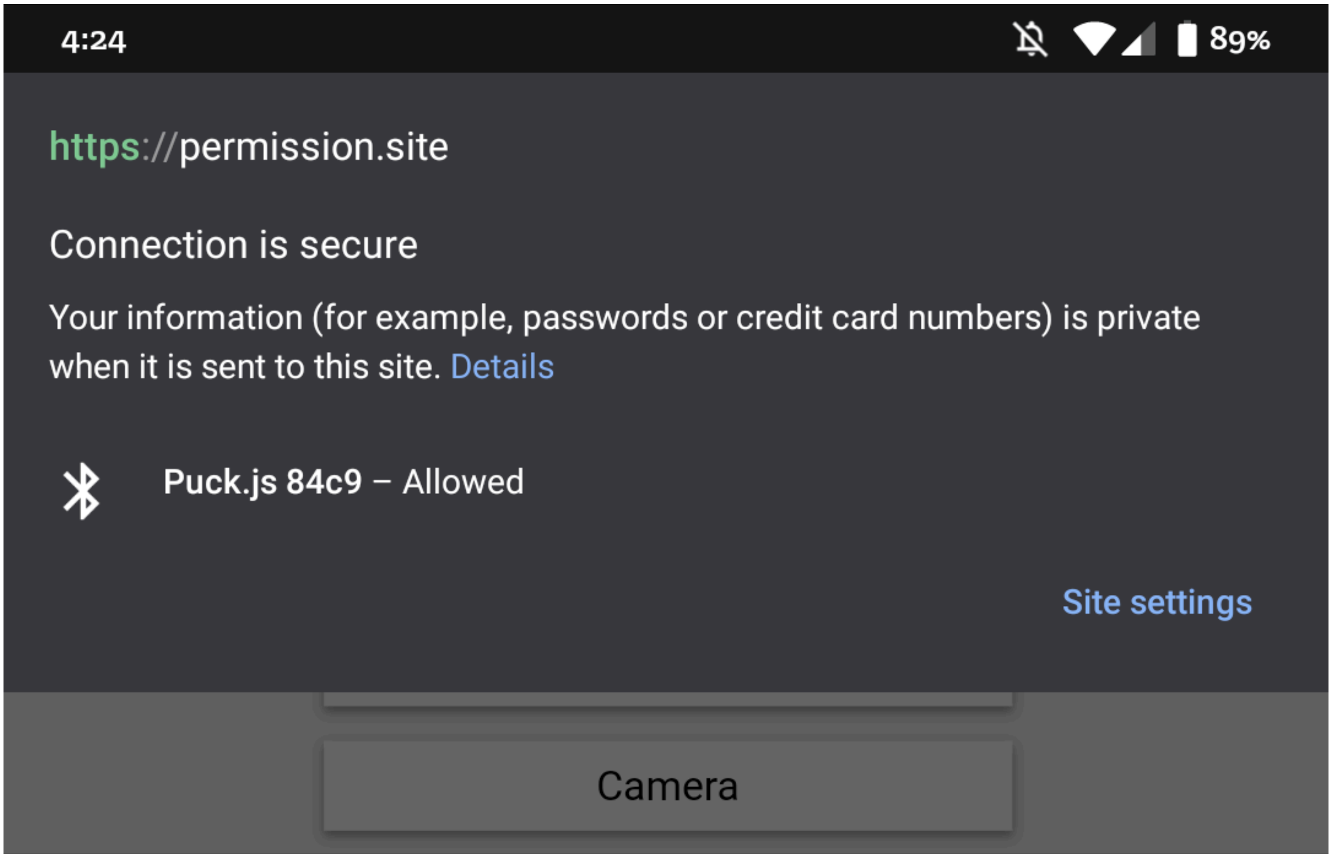
Bluetooth

Bluetooth
Block sites from connecting to devices

The first Bluetooth permissions screen when Web Bluetooth is blocked entirely.

Android Page Info

Page Info for Android is automatically implemented through the same code path as the desktop implementation.



4:24 89%

https://permission.site

Connection is secure

Your information (for example, passwords or credit card numbers) is private when it is sent to this site. Details

Puck.js 84c9 – Allowed

Site settings

Camera

The Page Info dialog displaying an entry for a granted Bluetooth device permission.

Android Scan Notification

When a scan is active as a result of a call to BluetoothDevice.watchAdvertisement(), Android needs to display a notification to inform the user that the site has started a scan. To implement this UI change, the [MediaCaptureNotificationService]() class can be used as a reference. A similar BluetoothNotificationService class can be created.

When a scan is started, WebContentsImpl::{In,De}crementBluetoothConnectedDeviceCount() is used to update the number of active scans. This method will trigger a tab invalidation notification, which will end up in [TabWebContentsDelegateAndroidImpl.navigationStateChanged()](). This is where the BluetoothNotificationService should add a notification if there is a scan active using WebContents::IsScanningForBluetoothDevices(). This method can be exposed to Java by using the WebContentsAndroid class, which wraps around WebContentsImpl.

Alternative designs

request() API

The `navigator.permissions.request()` API is non-standard, but it is defined in a separate specification, titled [Requesting Permissions](). This API won't be implemented for Web Bluetooth because it is non-standard.

The `navigator.permissions.request()` method in JS calls the `blink::Permissions::request()` method in C++. This method uses a helper method to convert the given `PermissionDescriptor` into a `PermissionDescriptorPtr` while filling in any extra data attached to the permission descriptor.

The `PermissionDescriptorPtr` is then passed to the `RequestPermission()` method of the `PermissionService`, which is an abstract class that is implemented by `PermissionServiceImpl`. The actual permission request is performed by the `RequestPermissions()` method, which processes a list of permissions, but only one is passed into this method by `RequestPermission()`. In this method, if the request is coming from a context where it's not possible to show a permission prompt, then `GetPermissionStatus()` is called, which is what the `query()` API does. If a permission prompt is able to be shown, then the list of permissions is processed by converting the `PermissionDescriptorPtr` to a `PermissionType`. The conversion is done with the `PermissionDescriptorToPermissionType()` method, so a `PermissionType` for Bluetooth will need to be added.

Check Chooser Permissions from PermissionServiceImpl

The `PermissionServiceImpl::RequestPermissions()` method is a great spot to diverge from the normal permissions request logic to perform chooser permissions request instead because it still contains the `PermissionDescriptor`. If the `PermissionDescriptor` belongs to Web Bluetooth, then the `WebBluetoothServiceImpl::RequestDevice()` method can be called with a `WebBluetoothRequestDeviceOptionsPtr` constructed from the `PermissionDescriptor` extension data.

When the `BluetoothChooserContext` is implemented, a specific `RequestChooserPermissions()` method can be created in the `PermissionServiceImpl` that can call the appropriate chooser context based on the `PermissionType`. A map of `PermissionType` to `ChooserContextBase*` can be done, similar to `kChooserTypeGroupNames[]` in [site_settings_helper.cc](), to use the appropriate chooser context.

Check Chooser Permissions from PermissionManager

Alternatively, the check for chooser permissions can be performed once the chrome layer is reached. To do this, the `PermissionDescriptor` will need plumbed all the way to

`PermissionManager::RequestPermissions()` in order to be able to use the extra Bluetooth request options attached to it. The `PermissionManager` needs to create a `BluetoothDeviceChooserController` (for other chooser APIs, it will need to show the appropriate chooser) and call `GetDevice()` on it with the request device options and callbacks for success and failure. The `BluetoothDelegate` class will need to provide a way for the `PermissionManager` to access the chooser controller, or an equivalent class needs to be implemented in the chrome layer.

Once a device is selected by the user, the chooser controller needs to be destroyed and the `BluetoothDevice` needs to be retrieved using the device address from the Bluetooth adapter class. Then a `WebBluetoothDeviceId` needs to be generated. Lastly, a `WebBluetoothDevice` needs to be created with the generated ID and the device name. The `BluetoothDelegate` will need to provide an interface for generating the ID, since `WebBluetoothDeviceId` is in the content layer.

Once the `WebBluetoothDevice` is created, the last step is to create the `BluetoothPermissionStatus` with `PermissionState = "granted"`, the `onchange EventHandler`, and the `devices` array populated with the granted device, and resolve the promise with this permission status.

Refactor navigator.bluetooth.requestDevice()

One support for the Permissions API has been implemented, the `navigator.bluetooth.requestDevice()` method can be updated to call `navigator.permissions.request()` internally, since both methods will essentially perform the same function. Eventually, the navigator.bluetooth.requestDevice() API should be deprecated in favor of the Permissions API.

Permission Storage

Storing Temporary Web Bluetooth Permissions

Alternatively, if `GrantServiceAccessPermission()` is called with `is_persistent` set to false, then the Bluetooth device permission is stored in maps within the class that are structured similarly to the maps in `BluetoothAllowedDevices`. These maps are cleared when the class is destroyed upon the closing of the browser.

The permissions for Bluetooth devices detected through `requestLEScan()` should be cleared when the browser is closed. Therefore, the `BluetoothChooserContext` can store these in maps within the class, which will be cleared when the class is destroyed upon the closing of the browser.

The `BluetoothChooserContext` returns `WebBluetoothDeviceIds` for devices with the `GetWebBluetoothDeviceId()` method. The method iterates over the object list to find a matching device address. If a match is found, the corresponding device ID is returned. If a match is not found, then the scanning maps are queried. If the maps have an entry for the device address, the ID value is returned. Otherwise, the device is new and an ID is generated for it. The address and ID are stored in the scanning maps to store their association temporarily.

When permission is granted to a site, the scanning maps are checked first for an existing device entry. If an entry exists, the ID is used when creating the permission object for the device. After the permission object is created and stored, the entry in the scanning map is cleared.

Deterministic ID Generation

With the current method being used to generate the Bluetooth ID, it is not possible to generate the same ID for the same device. In order to associate the generated ID to the device is via two maps. For devices added through the `requestDevice()` API, the size of these maps should be pretty low. However, through the `requestLEScan()` API, the size of these maps can become very large, depending on how many Bluetooth devices are around the user that are sending out advertisements. The Bluetooth devices detected by the Scanning API are not actually connectable, so an additional map is needed to specify which devices can be connected to. Using deterministic ID generation is not necessary for this feature, and it matters more for the Web Bluetooth Scanning API.

Instead, the method used to generate the Bluetooth ID can be changed to one that will be able to generate the same, unique ID for a given Bluetooth device. This would eliminate the need to keep maps to associate the ID and device address. However, to prevent sites from fingerprinting users, the generated ID should have the following properties:

- The Bluetooth device identifiers should not be able to be reversed engineered from the generated ID.
- The generated ID should be different across origins for the same Bluetooth device.
- The generated ID should be different for the same origin after a site permission reset.

The table below demonstrates the factors that determine whether a generated ID would be the same for a Bluetooth device or different.

Bluetooth Device IDs	Requesting and Embedding Origins	Permissions Cleared	Generated IDs Before and After
Same	Same	No	Same
Different	Same	No	Different
Same	Different	No	Different
Same	Same	Yes	Different

The HMAC algorithm using the SHA256 hash algorithm can provide an ID that fits these constraints, and it can be used to verify the integrity and authentication of the Bluetooth ID created from it. The `crypto::HMAC` class provides an implementation of this algorithm. The class must be initialized with a key, and this key can be unique for each pair of requesting and embedding origins. That way, the signed data produced by the algorithm is unique for each site for the same Bluetooth device.

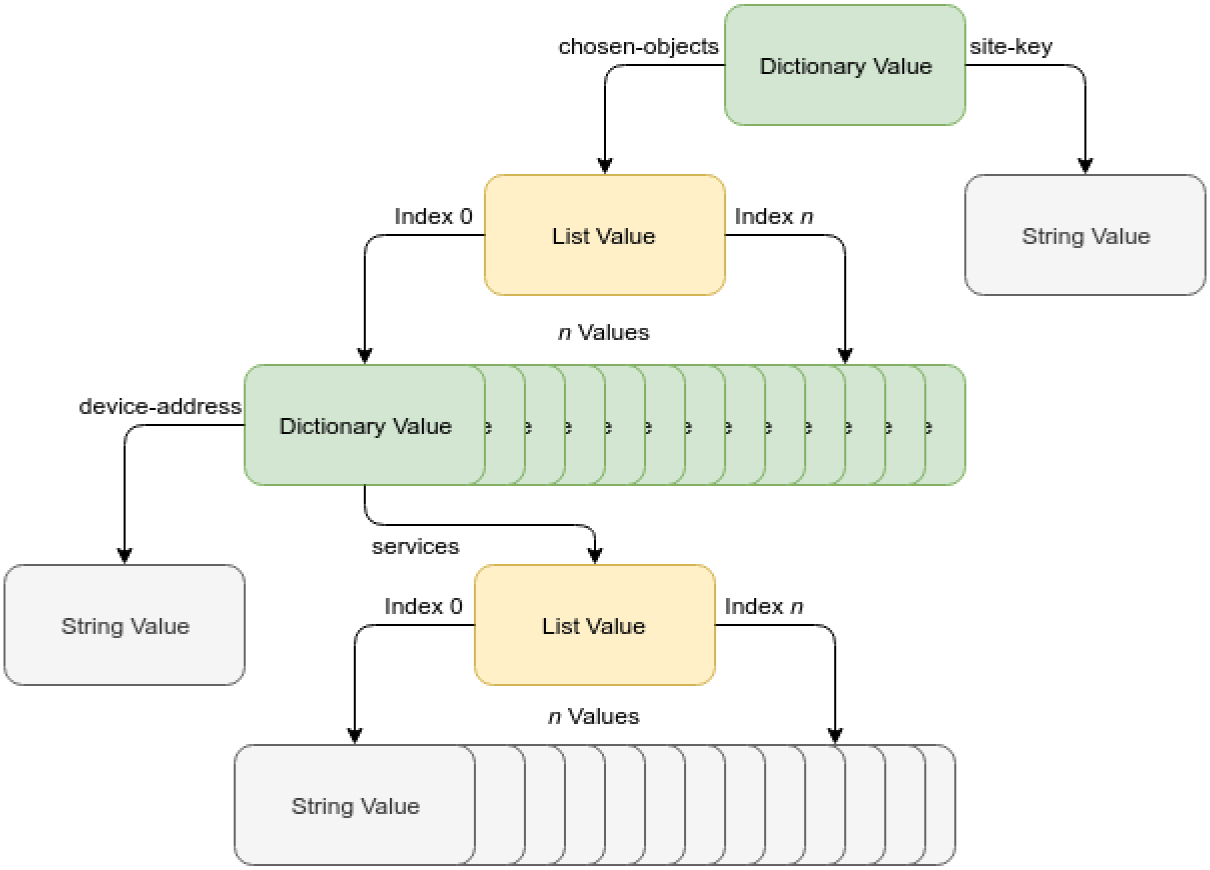
The key for the algorithm can be created with the `crypto::SymmetricKey` class using the `HMAC_SHA1` algorithm and stored in the user's settings. If the user resets a site's permissions, then the key will be destroyed so that a new one is created the next time that a Web Bluetooth permission is granted. The data that is signed by the HMAC algorithm will be a concatenation of the Bluetooth device identifier and the requesting and embedding origins that are requesting permission. To check if a Bluetooth device corresponds to ID, the `crypto::HMAC::Verify()` method can be used with the concatenated Bluetooth device identifier and origin data and the signed data.

Storing Web Bluetooth Permissions

`BluetoothAllowedDevices` contains several maps that map the following:

- device address string to `WebBluetoothDeviceId`
- `WebBluetoothDeviceId` to device address string
- `WebBluetoothDeviceId` to a set of `BluetoothUUIDs`
- `WebBluetoothDeviceId` to a boolean that is set to true if the device can be connected to.

Most of these maps are needed because the generated Bluetooth ID is completely random, so an association with the ID and device is needed. Additionally, the maps can contain devices that have been detected through the Web Bluetooth Scanning API, but haven't been allowed to be accessed by the site.



This diagram demonstrates the structure of the `base::Value` objects that will store Bluetooth permissions when the ID generation algorithm is deterministic.

With deterministic ID generation, only the `WebBluetoothDeviceId` to a set of `BluetoothUUIDs` will be needed. When `GrantServiceAccessPermission()` is called, a dictionary type `base::Value` object will be created to store the device permission. The object will contain a "device-address" key to store the device address and a "services" key to store the `BluetoothUUIDs`. The "services" key will contain another dictionary type object with a key for each `BluetoothUUID`, essentially making it a set. After the permission for the device is added using `ChooserContextBase::GrantObjectPermission()`, a `WebBluetoothDeviceId` will be generated to return. If this is the first time granting a Web Bluetooth permission for this pair of origins, then a new key needs to be generated to initialize the HMAC algorithm that will be used to create device IDs.

Checking Web Bluetooth Permissions

When checking Web Bluetooth permissions, the given `WebBluetoothDeviceId` from the content layer needs to be verified using `crypto::HMAC::Verify()` against all of the device addresses stored for that site. If the method returns true, then that means that the device is allowed to be used by the current origin.

Web Bluetooth may also require checking the services that are allowed to be used by the current origin. The allowed services are stored in the permissions for each Bluetooth device so this list can be iterated over to check if the service UUIDs match.

There is one case where a device address needs to be retrieved from a device ID in the Web Bluetooth service. In this case, the same logic to check the `WebBluetoothDeviceId` against the device address stored in permissions can be done to find a match. When a match is found, the device address can simply be returned.

Bluetooth Chooser Prompt

The Bluetooth chooser prompt needs to be able to give the user the option to grant Bluetooth device permissions once until the browser is closed or persistently across browsing sessions.

Developer requests

This is a list of developers who have expressed interest in having a feature like this or had problems that can potentially be addressed by this feature.

- https://bugs.chromium.org/p/chromium/issues/detail?id=974879#c1
- https://bugs.chromium.org/p/chromium/issues/detail?id=577953#c13
- https://github.com/WebBluetoothCG/web-bluetooth/issues/211#issue-130699789
- https://github.com/WebBluetoothCG/web-bluetooth/issues/365#issue-225334091
- https://github.com/WebBluetoothCG/web-bluetooth/issues/411
- https://github.com/WebBluetoothCG/web-bluetooth/issues/358
- https://github.com/WebBluetoothCG/web-bluetooth/issues/31#issuecomment-343919299
- https://stackoverflow.com/questions/45467214/is-it-possible-to-persist-a-bluetooth-le-connection-on-browser-refresh
- https://stackoverflow.com/questions/60604388/web-bluetooth-get-paired-devices-list
- https://stackoverflow.com/questions/60603666/web-bluetooth-bypass-pairing-screen-for-a-known-device-id
- https://stackoverflow.com/questions/59077656/automate-connecting-to-bluetooth-devices-from-chrome
- https://stackoverflow.com/questions/55531254/web-bluetooth-bypass-pairing-screen
- https://stackoverflow.com/questions/53467676/remembering-the-device-and-reconnecting-to-it
- https://stackoverflow.com/questions/43633589/web-bluetooth-api-store-connection-object

Metrics

Success metrics

Success for this feature can be measured as follows:

1. Sites are able to get a list of all Bluetooth devices that it has permission to access regardless of whether they are currently connected or not.
2. Users are able to use a Bluetooth device that they paired with a site without needing to grant permission to the site again. This should happen in the following scenarios:
    a. A Bluetooth device that has gone out of range/powered off goes back into range of the Bluetooth adapter.
    b. A Bluetooth adapter that has been powered off/removed is powered back on.
    c. The user closes the browser/tab and returns to the website at a later time.
3. Users are able to see all of the sites that have been granted permission to use a Bluetooth device and manage these permissions.

Regression metrics

Regression for this feature can be measured as follows:

1. The Web Bluetooth API no longer works as it did before the feature.
2. Sites see Bluetooth devices that they don't have permission to access or don't see Bluetooth devices that they do have permission to access.
3. Bluetooth devices that have gone out of range and back into range are not able to be used properly.
4. Bluetooth permissions are not displayed properly or are not able to be manipulated by the user.

Rollout plan

Waterfall

Core principle considerations

Security

This feature will allow the Web Bluetooth permission model to be consistent with how the permissions are done for WebUSB. A site will only be able to use a Bluetooth device if the user has allowed the site to access the device through a chooser prompt. In addition, this feature will allow the user to manage the permissions that have been granted for Bluetooth devices, allowing them more control over Bluetooth device access than was previously available.

Privacy considerations

This feature will enable websites to get a list of permitted Bluetooth devices using `navigator.permissions.query()`. These devices include ones that are not currently in the range of the adapter. In order for a device to be included in this list, the user needs to explicitly grant the site permission to use the device. A random device ID is generated for each device and for each site when the device permission is granted, and the ID is cleared when the permission is reset. Therefore, it would be difficult to fingerprint a user with this API since the site needs to have permission for the device before they can see it and the random IDs prevent devices from being able to be tracked across sites.

As mentioned in the Core principle considerations section, users will now have greater control over the Bluetooth devices permissions that have been granted to sites. They will be able to see all of the Bluetooth devices and the corresponding sites that have permission to use the device, and revoke any permissions that they choose. These permissions will be visible in Site Settings for desktop and Android, as well as the Page Info dialog box. Device permissions granted in incognito mode are revoked upon the destruction of the off the record profile.

Testing plan

The implementation of the Fake Bluetooth scanning API should be implemented in order to create web platform tests that ensure the consistent behavior of reconnecting Bluetooth devices.

Implementation plan

These are the tasks required to implement this feature:

Task	**Bug**	**Progress**
Get permitted devices.	577953	Implemented behind #enable-experimental-web-platform-features flag.
Refactor permissions backend.	589228	Implemented behind #web-bluetooth-new-permissions-backend flag.
BluetoothDevice.watchAdvertisements().	681435	WIP behind #enable-experimental-web-platform-features flag.

Desktop Site Settings	563724 601523	Implemented behind #web-bluetooth-new-permissions-backend flag.
Desktop Page Info	689240	Implemented behind #web-bluetooth-new-permissions-backend flag.
Android Site Settings	659337	WIP behind #web-bluetooth-new-permissions-backend flag.
Android Page Info	659337	Implemented behind #web-bluetooth-new-permissions-backend flag.
Update connect to use platform APIs to get device or discover device.	681435	Not started. Best case: 2 weeks Worst case: 8 weeks

Web Bluetooth API status for each operating system:

API	**Windows**	**macOS**	**Linux**	**Android**	**ChromeOS**
getDevices()	Working	Working	TBD	Working	TBD
watchAdvertisements()	WIP	Working	TBD	Working	TBD