API proposal

User Script Support in Scripting API

Author	Emilia Paz (Google)
Reviewers	Devlin Cronin (Google), Simeon Vincent (Google), ...

Note for contributors: when identifying issues or bugs, please include how the same is accomplished in an MV2 extension. The goal is to achieve functional parity with MV2 for user scripts managers.

Background

Manifest Version 3 (MV3) restricts the ability for extensions to use remotely-hosted code since it is a major security risk (code that isn't in the extension package cannot be reviewed or audited). This is directly at odds with user script managers, which fundamentally require the execution of arbitrary code (the user scripts themselves). This creates a feature gap between MV2 and MV3.

In MV3, extensions can use the chrome.scripting API to dynamically execute scripts, either entirely programmatically at runtime using <global>.scripting.executeScript() or by registering a content script dynamically with <global>.scripting.registerContentScripts(). However, neither of these allow for code outside of the extensions package:

- `<global>.scripting.executeScript()` can take either a set of files from the extension's package or a Javascript function (with optional curried arguments).
- `<global>.scripting.registerContentScript()` takes a set of files.

Proposal

User scripts satisfy an important use case for power users, allowing them to quickly and easily tweak the functionality of a web page. Therefore, we propose to **allow registering content scripts with arbitrary code by adding new functionality to the scripting API**. However, we will also take a number of steps to help reduce the abuse-ability of this API and ensure we don't lose the benefits of the remotely-hosted code restrictions from MV3.

From this point on, content scripts with arbitrary code will be referenced as “user scripts” since this is the only scenario where this should be supported.

Goal

- Achieve functional parity with MV2 for user script managers.

Abuse Scenarios Mitigations

Execution World Restriction

User scripts will only be allowed to execute in the main world or a new UserScriptWorld. Specifically, user scripts should not be allowed to access the extension’s isolated worlds and APIs.

USER_SCRIPT World

- Isolated from the web page (similar to other isolated worlds), but will be un-permissioned. It will not have access to any extension APIs or cross-origin exceptions (these don't exist in Chrome, but do in other browsers). However, it will be exempt from the page's (and extension's) CSP.
- Share DOM with the web page. Code from both worlds cannot directly interact with each other, except through DOM APIs.
- Can communicate with different JS worlds via window.postMessage(). In the future, a dedicated API to communicate between worlds could be considered.
- When an asymmetric security relationship may exist, the MAIN world is considered to be less privileged than the USERSCRIPT world

Permissions

A new extension permission will be added, and will be required in order to register user scripts. This permission should be scoped specifically to the purpose of user scripts (as opposed to a general "remotely-hosted code" permission).

Browser vendors then can use it to:

a) Inform their review pipeline that this is a particularly risky extension, and extra caution should be taken to ensure this is a valid use of the API.

b) Reduce the likelihood of extensions using it for other purposes and allows browser vendors to act on any permissions that do as being non-compliant

c) Present the user with different permission warnings, UI, or other gating, if they deem it necessary.

API Schema

Types

```
dictionary RegisteredUserScript {
```

```
  boolean? allFrames;
  string code;
  string[]? excludeMatches;
  string id;
  string[]? matches;
  runAt runAt;
  executionWorld world;
}
```

```
dictionary ScriptFilter {
  string[]? ids;
}
```

Methods

```
chrome.scripting.registerUserScripts(
  scripts: RegisteredUserScript[],
  callback?: function
)

chrome.scripting.unregisterUserScripts(
  filter?: ScriptFilter[],
  callback?: function
)
chrome.scripting.getRegisteredUserScripts(
  filter?: ScriptFilter[],
  callback?: function
)

chrome.scripting.updateUserScripts(
  scripts?: RegisteredUserScript[],
  callback?: function
)
```

Having different types and methods than content scripts is beneficial in the following ways:

a. Better documentation. The clear separation between user and content scripts will reduce confusion for developers for which API methods to use and what the capabilities / restrictions of each are. This naming also more clearly communicates that this capability is not meant as a general purpose way to inject arbitrary scripts.

b. Stronger enforcement of remote host code abuse. Reviewers can see in a more clear way where user scripts are being used to spot abuse patterns.

c. Easier engineering implementation. Browser vendors should be able to restrict the execution world and different API methods behind features. Also, by not sharing the same methods as content script there would be no need to handle conditional parameters (e.g. if code parameter is set, world cannot be set to the extension's isolated world).

Other considerations

- When multiple scripts match and have the same runAt schedule, scripts targeting ISOLATED are executed before USERSCRIPT. This allows the extension to prepare the execution environment before the user script code is executed.
- User scripts are always persisted across sessions, since the opposite behavior would be uncommon,

Browser level restrictions

From here, each browser vendor should be able to implement their own restrictions. Chrome is exploring limiting the access to this API when the user has enabled developer mode (bug), but permission grants are outside of the scope of this API proposal.

Alternatives Considered

Generally allow remotely-hosted code

Instead of having a special carve out for user scripts (with a new permission), we could allow extensions to use remotely-hosted code generally. We find this an unacceptable security risk to our users.

Don't support user script managers

User script managers are, in a way, an extension system unto themselves. They allow the addition and management of various third-party-developed functionalities that change the user's browsing experience. Instead of supporting user script managers, we could require that all user scripts be turned into extensions directly.

However, this would be an undue amount of burden for users. Many users may have dozens or even hundreds of user scripts installed. Given our simultaneous efforts to increase the visibility of extensions, requiring each of these scripts to be a full extension would result in an incredibly cluttered and clunky UI.

Firefox's userScript API

Firefox introduced a userScripts API in Manifest V2 that allowed extensions to register dynamic user scripts, including the ability to execute arbitrary code. This API works by returning a reference to a `RegisteredUserScript` object in response to the `userScripts.register()` API call. This `RegisteredUserScript` can be used to unregister the script, and the script will be automatically unregistered if this object is garbage collected.

The main issue with this API is that it is fundamentally incompatible with any ephemeral background context (either event pages or service workers). When the background context is torn down, the user script would be unregistered.

Add functionality to Scripting API "content scripts" methods

`RegisteredContentScript` is used by multiple methods to register, unregister, update and get content scripts. Currently only js and css files are supported. We can add a new code field that takes a string with the user script.

```
dictionary RegisteredContentScript {
  ...
  string? code;
  ...
}
```

The following methods would need to be updated:

- `<global>.scripting.registerContentScripts()`: Add new optional field code. When code is present js and css cannot be set because user scripts run in separate worlds. Therefore, executionWorld cannot be set and would default to USER_SCRIPT. If the user tries to set any of these, an error message would return.
- `<global>.scripting.unregisterContentScripts()`: no changes since user scripts would be removed the same way
- `<global>.scripting.updateContentScripts()`: support updating user scripts
- `<global>.scripting.getRegisteredContentScripts()`: no changes since it returns scripts including user scripts.

However, having content scripts and user scripts under the same methods would be confusing for the developer, and more difficult in the implementation side.

Future Enhancements

USER_SCRIPT World Communication

In the future, we may want to provide a more straightforward path for communication with a USER_SCRIPT world (as opposed to the `postMessage()` approach highlighted above).

Custom CSP

We could allow the user script registration to specify its own content security policy, allowing user script to opt-in to a more secure world.

Separate USER_SCRIPT Worlds

In addition to specifying the execution world of `USER_SCRIPT`, we could allow extensions to provide an identifier for the world. Scripts injected with the same identifier would inject in the same world, while scripts with different world identifiers inject in different worlds. This would allow for greater isolation between user scripts (if, for instance, the user had multiple unrelated user scripts injecting on the same page).

Open Questions

Feel free to highlight any open questions here.