mozilla Labs

Jetpack

Improving Security in Firefox Add-ons

Brian Warner, Mozilla Labs
QCon 2010, San Francisco

slides: http://people.mozilla.com/~bwarner/qcon10/

Overview

- Why add-ons are important
- Add-on security issues
- Mozilla Add-ons and Jetpack
- Jetpack Security
- Review-based Security Analysis

mozilla LABS
Jetpack

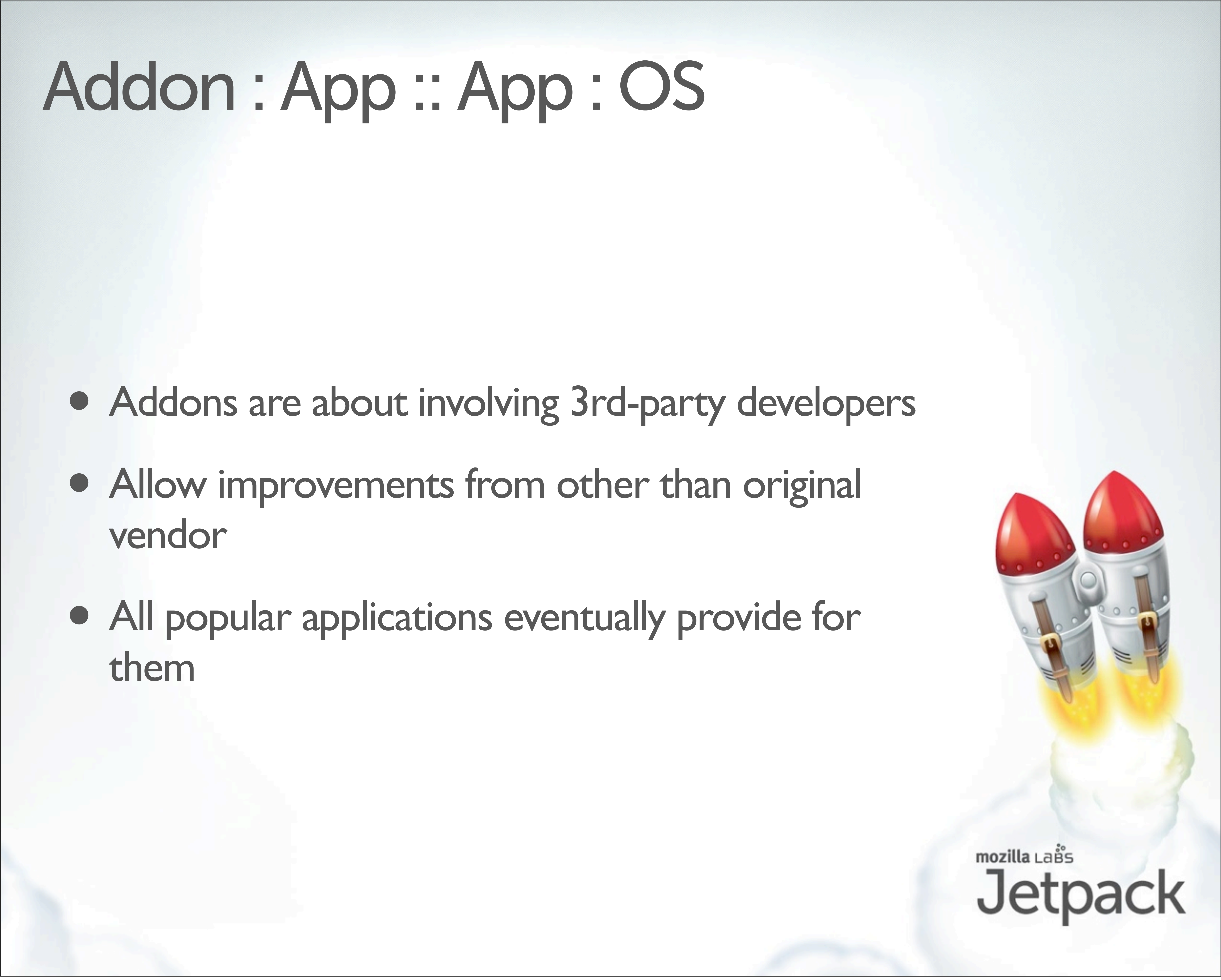
Takehome Messages

- Web browsers are increasingly fundamental tools, so add-ons are security-critical
- For users, software install is an economic tradeoff: risk-vs-reward, difficult when both are unknowns
  - Also a negotiation between user and vendor, mediated by user agent
- Users should be in control, but deserve help: delegate and distribute the decision-making

mozilla LABS

Jetpack

This is a somewhat philosophical talk. The deep ideas that I want you to walk away with are these..



Addon : App :: App : OS

- Addons are about involving 3rd-party developers
- Allow improvements from other than original vendor
- All popular applications eventually provide for them

audience survey: addon usage, addon developers

Desktop applications, web browsers, online games.. all big applications, once they get popular enough, succumb to user pressure to add extensibility.

Firefox add-ons

- Very popular: 2.1 billion downloaded so far, 170M active daily. Roughly 10K in AMO gallery, about 300 created each month.
- big reason for users to prefer Firefox, big reason to avoid upgrades

All the major browsers provide for add-ons. Safari added support this summer, and Opera released some alphas last week with extension support.

Why Browser add-ons matter

- The web browser is becoming the new OS
- It mediates most of our interaction with the Internet
  - Other protocols and UIs are being replaced
- They add 3rd/4th/Nth parties to the usual vendor/user equation: competing interests and incentives
- Addons modify a user's experience with an application. Browser addons modify a user's experience with the whole Internet.

Add-on APIs

- some apps are carefully built to support addons
- the language in which addons are written, and the API to which it speaks, dictates its abilities
  - low-level languages make everything possible, including bugs and vulnerabilities
  - libraries make some things easy
  - high-level languages/interfaces make specific things possible, disallowing the rest
    - not as flexible, but generally safer

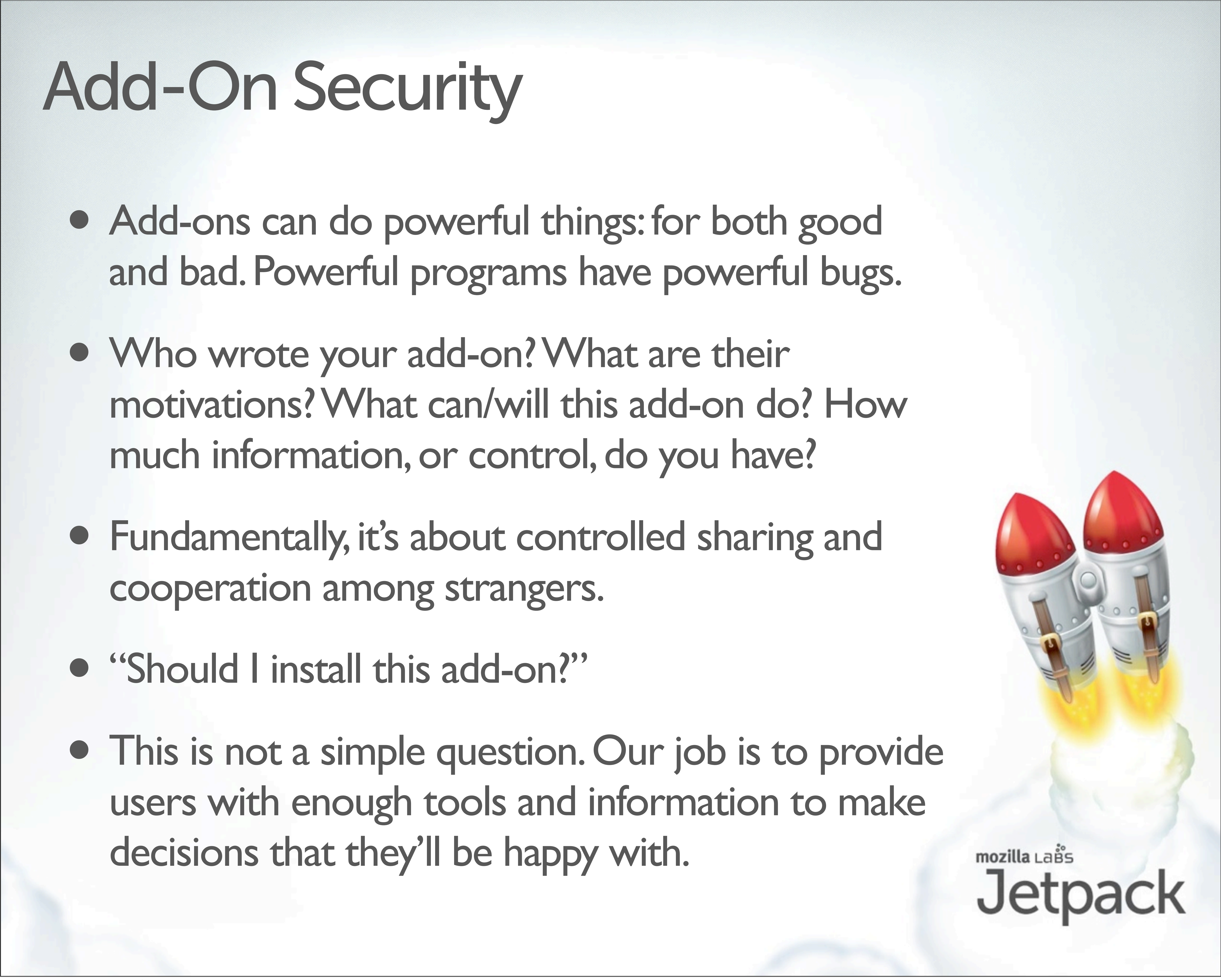
mozilla LABS

Jetpack

Browser Add-Ons: JS+HTML

- trend in browser addons is to "Build them out of the Web": Javascript, HTML, CSS
- Makes add-on development accessible to webdevs, not just browserdevs
- JS is fairly high-level, can be confined: allow for a good variety of API possibilities
  - (if add-ons were x86 machine code, confinement options would be worse)

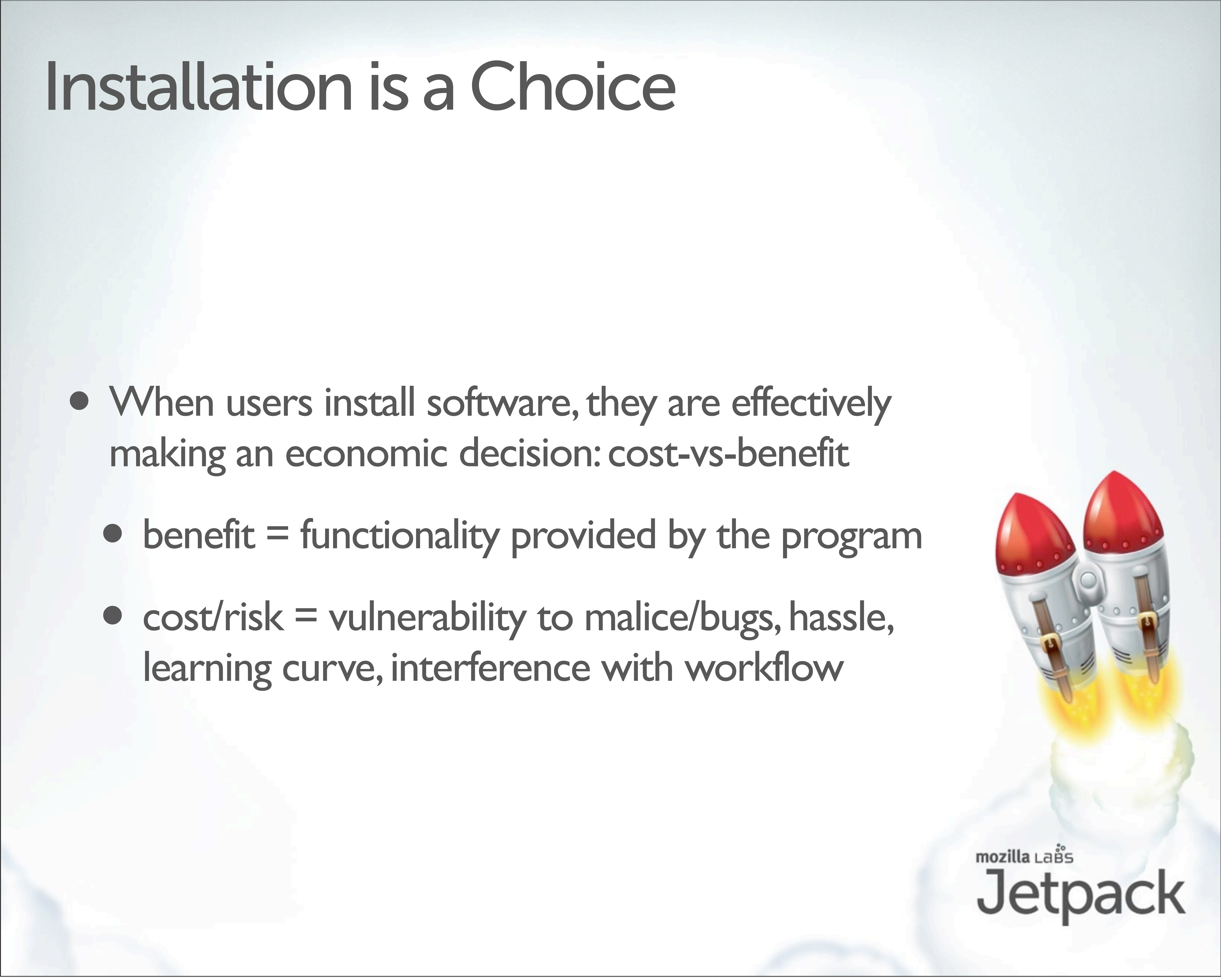
Add-On Security



Add-On Security

- Add-ons can do powerful things: for both good and bad. Powerful programs have powerful bugs.
- Who wrote your add-on? What are their motivations? What can/will this add-on do? How much information, or control, do you have?
- Fundamentally, it's about controlled sharing and cooperation among strangers.
- "Should I install this add-on?"
- This is not a simple question. Our job is to provide users with enough tools and information to make decisions that they'll be happy with.

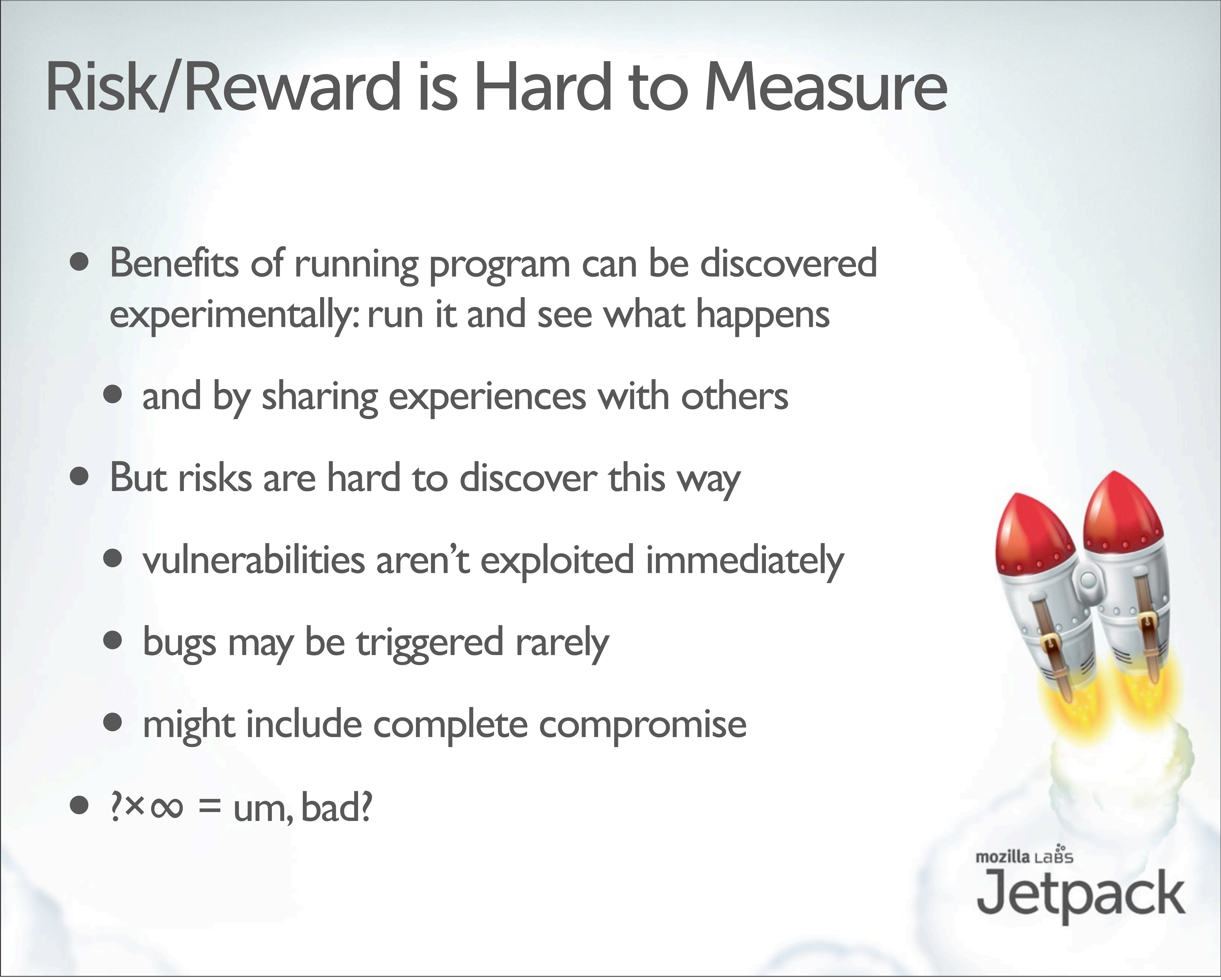
The power of an add-on program is, at first glance, limited by the language in which it is written and the API it can call. This is actually the same situation we have for regular desktop applications, except we aren't used to thinking of limited-power apps (with the growing exception of iphone/android apps).



Installation is a Choice

- When users install software, they are effectively making an economic decision: cost-vs-benefit
  - benefit = functionality provided by the program
  - cost/risk = vulnerability to malice/bugs, hassle, learning curve, interference with workflow

They may not realize it, but users are making a deliberate cost-benefit analysis when they install software of any sort. On the plus side, they might get to play a game, or find lower prices, or whatever. On the down side, they might also have to learn something new, or lose some other feature that they liked, or have all their secrets published to 4chan and then their drive gets wiped.

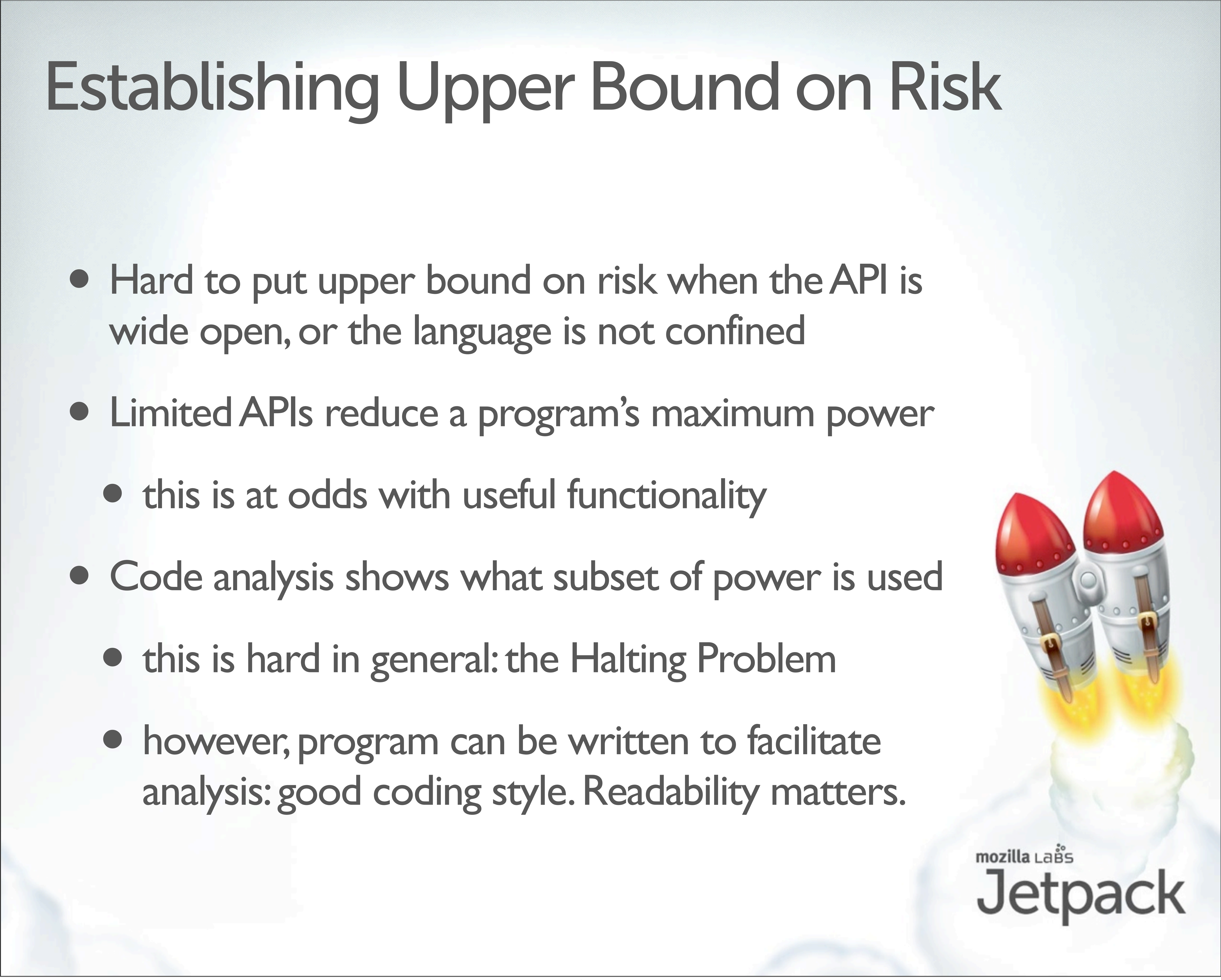


Risk/Reward is Hard to Measure

- Benefits of running program can be discovered experimentally: run it and see what happens
  - and by sharing experiences with others
- But risks are hard to discover this way
  - vulnerabilities aren't exploited immediately
  - bugs may be triggered rarely
  - might include complete compromise
- ?×∞ = um, bad?

We're often willing to extend credit to the program: we'll spend the time installing it in the hopes that it will do something useful for us, especially because that's usually the best way to find out what it does. We also look for reviews and places where we can leverage the experiences of other users. But the costs are hard to evaluate this way, especially the security risks: they happen infrequently enough that you probably won't discover them in a casual walkthrough. And if the risk includes complete compromise, the cost may be tremendous: bank accounts, personal data, etc.

So the "cost" in this analysis is an unknown probability of compromise times infinity dollars. How do you work with that? Users could make a better cost-benefit decision if we could somehow limit the downside.



Establishing Upper Bound on Risk

- Hard to put upper bound on risk when the API is wide open, or the language is not confined
- Limited APIs reduce a program's maximum power
  - this is at odds with useful functionality
- Code analysis shows what subset of power is used
  - this is hard in general: the Halting Problem
  - however, program can be written to facilitate analysis: good coding style. Readability matters.

mozilla Labs
Jetpack

a typical desktop program runs with the full authority of the user account: it can delete any file, read secret data, talk to any server. Add-ons which run with full power can do anything the program can do.

But it's increasingly common to write addons in a language that can be confined, and to give them APIs that are less powerful than the whole program. This starts to bring down the upper bound: the addon can only do things that the API provides, no more. This reduces the utility of the system: addons cannot do interesting, useful things that the API designer didn't provide for, which is the sort of limitation that motivated addons in the first place. So there's a pressure to give addons as much power as possible, but there's also pressure to give them very little power.

You may also be able to reduce the risk by analyzing the addon carefully, and trying to understand what it will and won't do by just looking at the code and predicting its future behavior. This is always labor-intensive, generally requires a skilled human, and actually technically impossible in the general case (otherwise we could solve the Halting Problem). But with enough effort, you may be able to develop some amount of confidence in your predictions. The problem is eased somewhat if you can require the author to write the code in a way that is easier to read, but that's not usually the case.

Power vs Safety

- We want powerful programs
- We want safe programs
  - Weak programs are safe, but boring
- My favorites:
  - isolated modules
  - POLA: Principle Of Least Authority
  - object capabilities

mozilla Labs
Jetpack

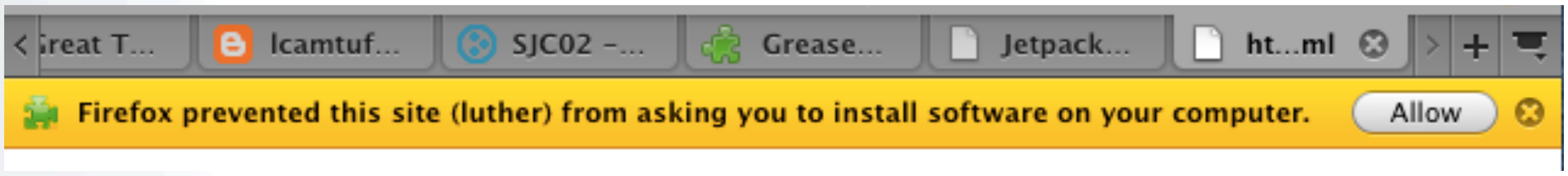
So we want powerful programs, but we also want safe programs, and weak programs are safe but boring. In general, the way forward is to break the program up into isolated modules, and apply the Principle Of Least Authority to each one. Give each component of the program as little power as possible. When possible, you narrow down each module to a single object, giving you the object-capability model.

In this model, you don’t try to read or reason about code that you get from external parties: you just run it in an environment where it can only access the things you’re willing to let it control. Reasoning about what arbitrary code is going to do, and in particular what it *won’t* do, is hard, and is only feasible for fairly simple programs that are voluntarily written to make such analysis possible.

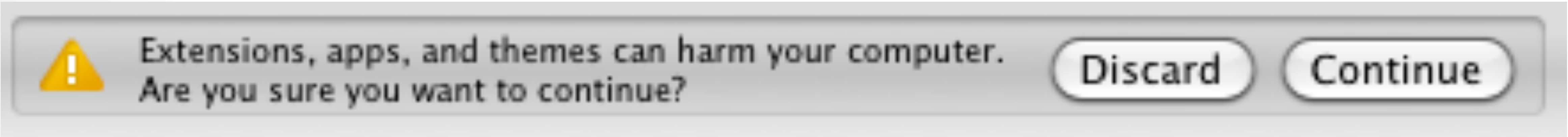
There’s an interesting tension here. At Mozilla, we’re dedicated to giving users control over their browser, which means giving them power. But most of those users also expect us to provide for their online safety. Meeting both expectations is a tricky process.

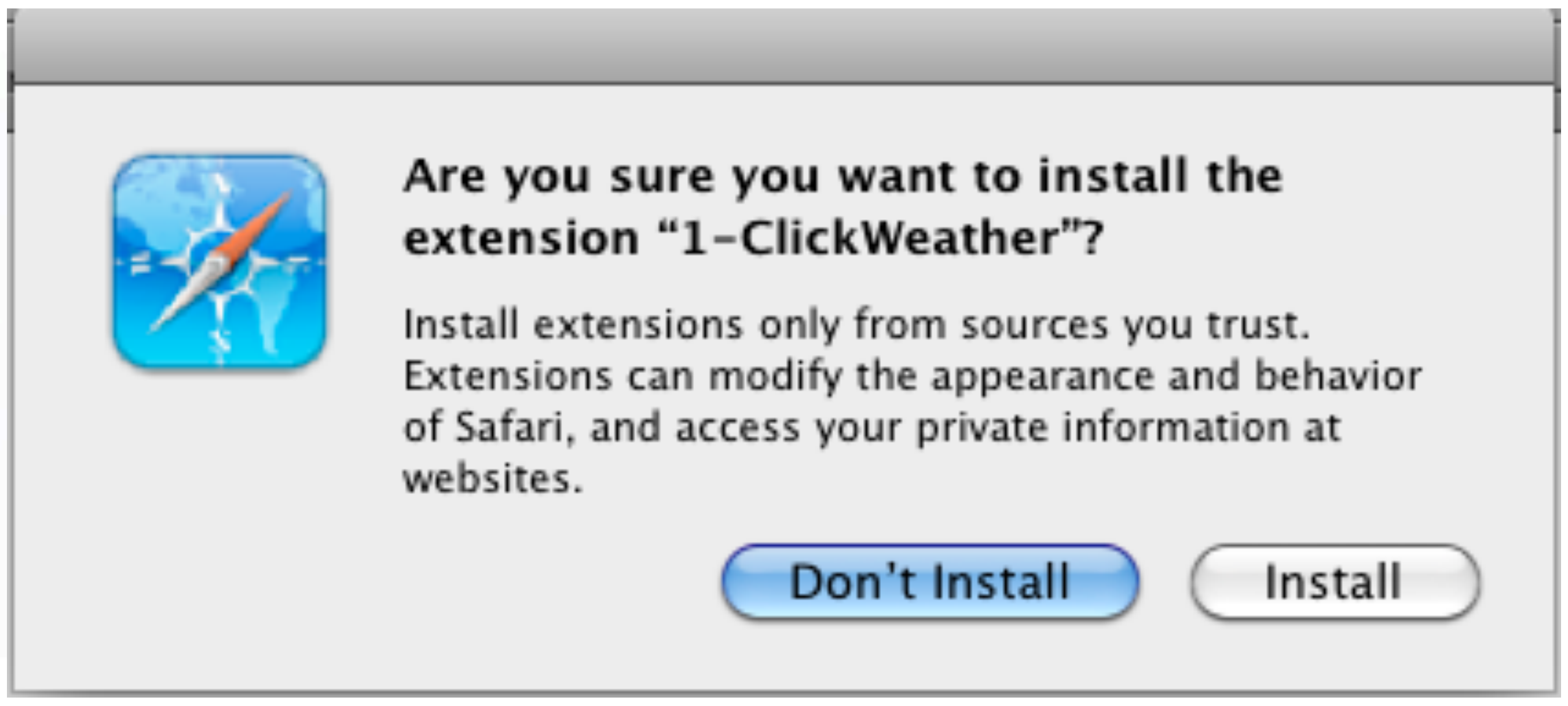
Installing Add-Ons

- User clicks a link to install an add-on
- Official galleries are whitelisted: fewer warnings
- Add-ons hosted elsewhere get scary warnings



Firefox prevented this site (luther) from asking you to install software on your computer.


Extensions, apps, and themes can harm your computer.
Are you sure you want to continue?

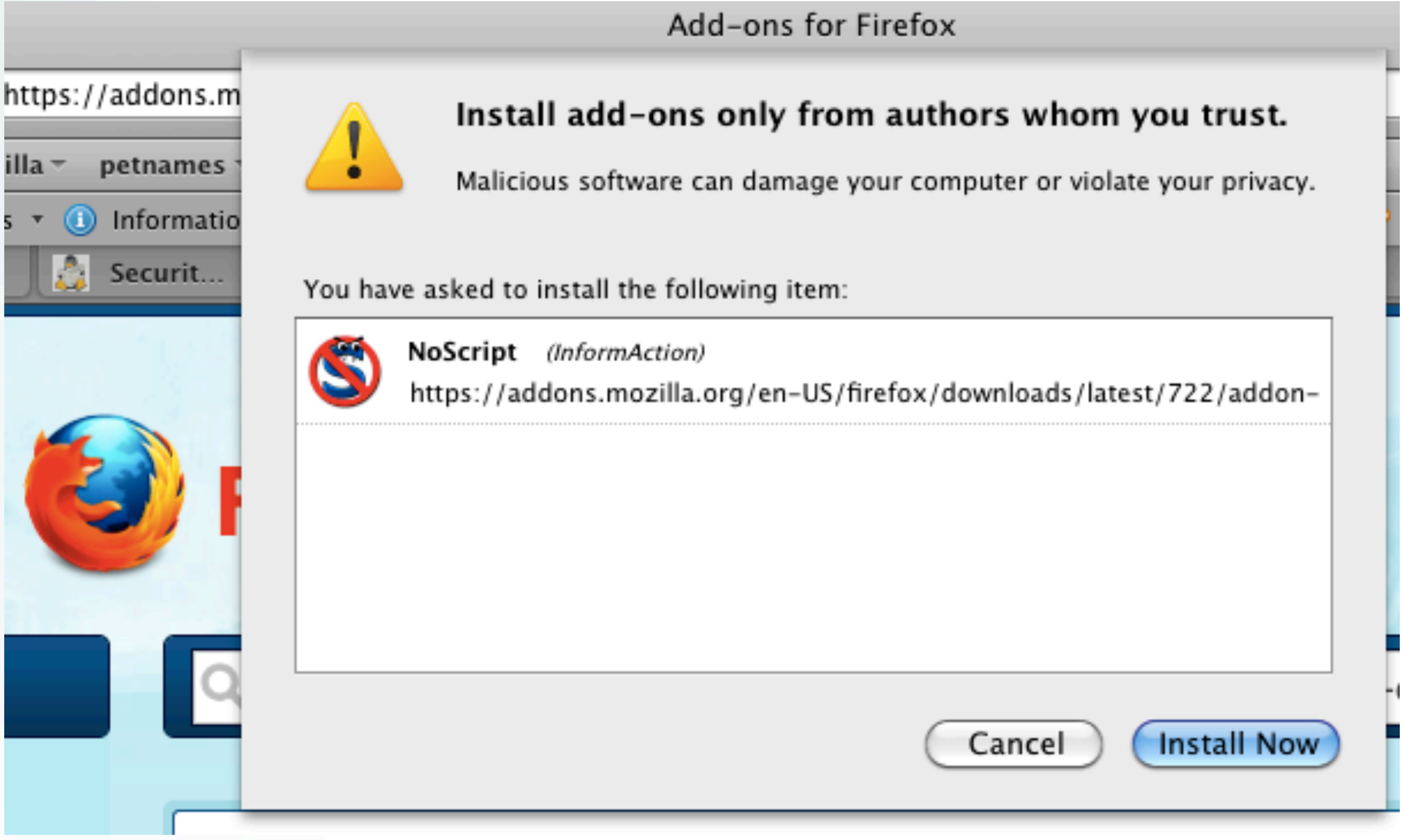

Are you sure you want to install the extension “1-ClickWeather”?

Install extensions only from sources you trust. Extensions can modify the appearance and behavior of Safari, and access your private information at websites.

Some security properties show up during the add-on installation process.
The major browser vendors all have official galleries where they promote certain extensions, and the browsers themselves whitelist these galleries to reduce the installation barrier. This adds a minor “speed bump” to addons that come from external sites (and thus we don’t know what kind of review they’ve been through). This is enough to encourage authors to publish their extensions through the official galleries, but not always enough to discourage users from installing unreviewed addons. SSL certificate warnings suffer the same problem.

Ask The User

- "Install?: Yes/No"
- like all security "monolog" boxes, not much of a decision



Add-ons for Firefox

Install add-ons only from authors whom you trust.

Malicious software can damage your computer or violate your privacy.

You have asked to install the following item:

NoScript *(InformAction)*
https://addons.mozilla.org/en-US/firefox/downloads/latest/722/addon-

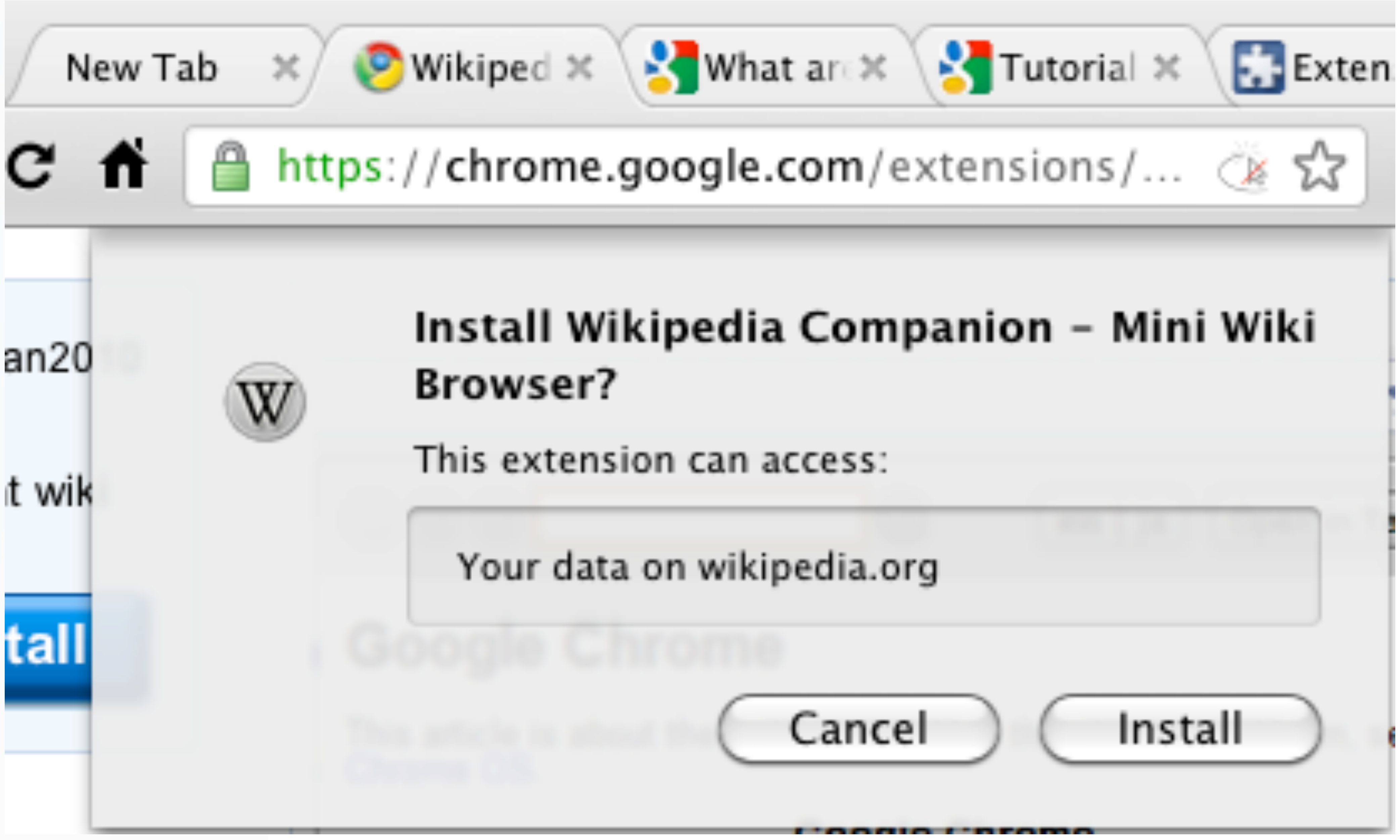
Addons, even from a whitelisted site, require user confirmation before installation (except for Safari, which really is 1-click). The Firefox install dialog actually imposes a 5 second delay before accepting confirmation, to reinforce the seriousness of the question it is asking.

This dialog box, which we might as well call a "monolog" box because it's really pretty one-sided, bugs me a lot, even though it's the best that firefox can do under the circumstances. The user is being asked to accept an unknown risk, in the hopes of an unknown benefit, and the only information they receive to distinguish this questionmark-vs-questionmark decision against any other one is the icon chosen by the addon author and the first three inches of a random URL. They're being asked to accept responsibility for arbitrary badness, encouraged to think about whether they trust the author (which isn't displayed, even if that could be done meaningfully). And trust to do what, exactly?

If you saw Marc Steigler's presentation earlier today, you may recognize this as an example of the "Blame The Victim" pattern.

Ask The User

- need to give useful information, then ask useful questions. What if we could:
  - show list of powers that program wants to use?
  - let user withhold specific authorities, or replace with stubs?
- larger authorities deserve larger knobs



Install Wikipedia Companion - Mini Wiki Browser?

This extension can access:

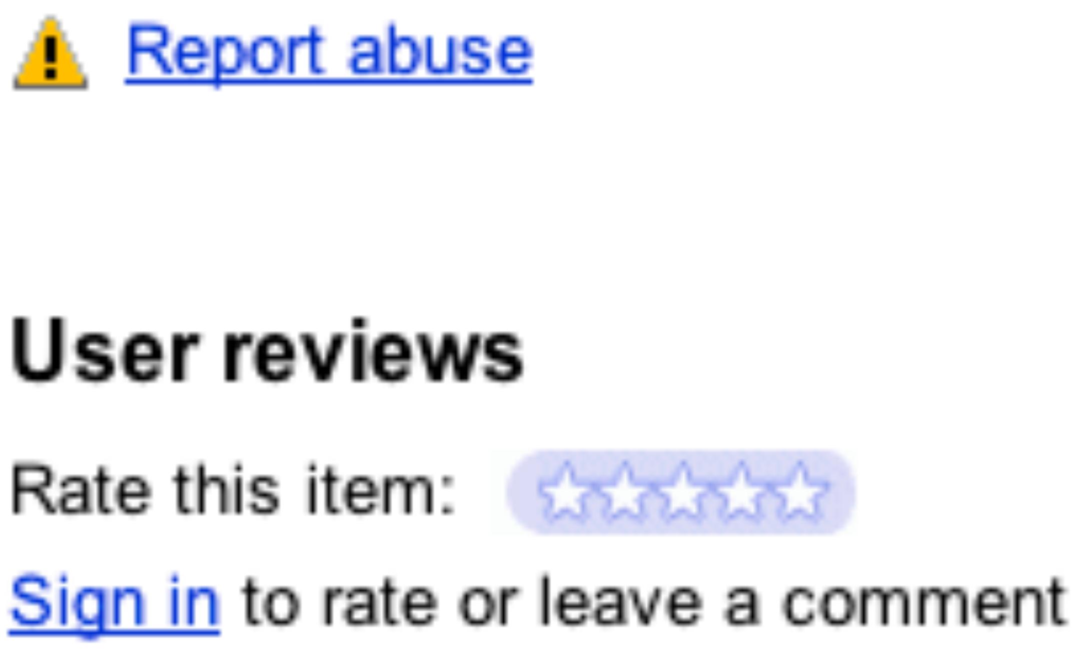
Your data on wikipedia.org

I like the direction Chrome is going: at least it is trying to provide the user with some information about the potential risks. But I'd like to see something better: giving the user more say in the way the new add-on code gets to interact with their browser. I'd like to see checkboxes on each authority, which can be withheld or replaced with fake stubs (think geolocation, or persistent storage which is actually flushed when the tab is closed). And bigger authorities (like passwords) should get visibly larger checkboxes, so the user doesn't give away their bank password as casually as they give away their email address.

The challenge, of course, is to involve the user only to the extent that they are interested and capable of participating. The decision to spend time understanding what powers to give the add-on is an economic one too, and most users quite reasonably do not want to spend that time. We should accomodate them, by letting them leverage other people's efforts (by getting advice from, and perhaps delegating their choice to, trusted reviewers, friends, and the general public).

Ex Post Facto Remedies

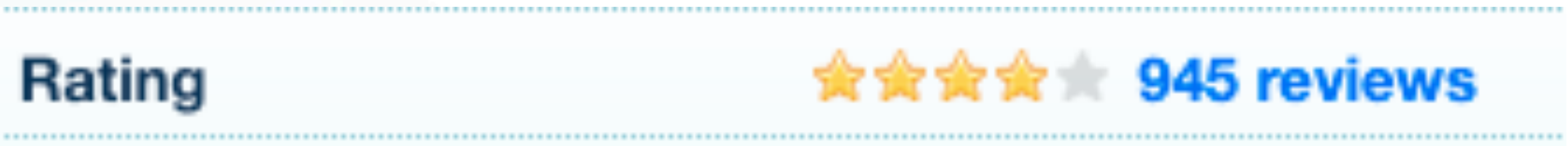
- Reputation, Ratings, Revocation
  - learn from the suffering of others
- fuzzy, social, but effective


⚠ Report abuse

User reviews

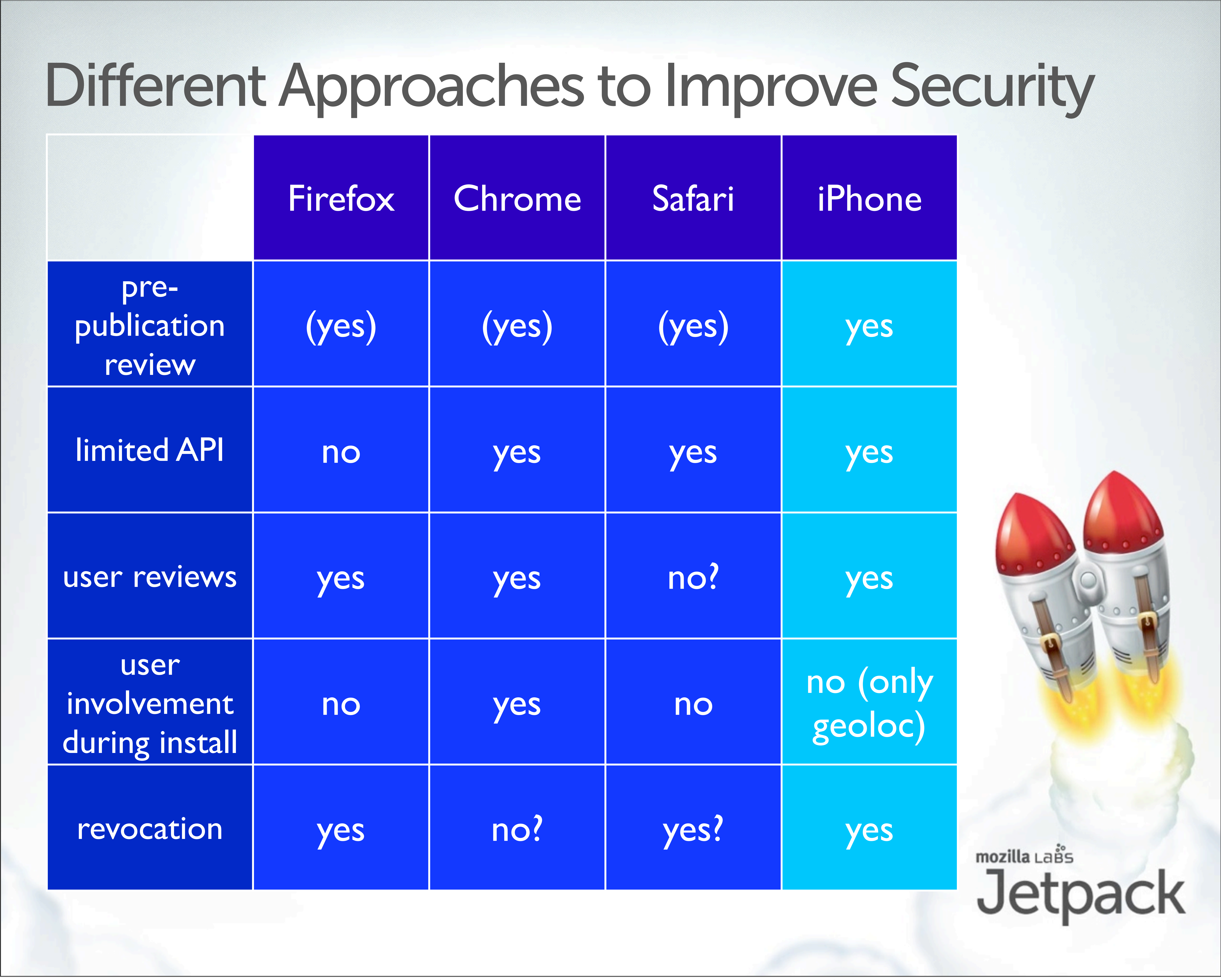
Rate this item: ☆☆☆☆☆

Sign in to rate or leave a comment


Rating ★★★★☆ 945 reviews

mozilla Labs
Jetpack

Finally, most add-on systems can survive the lack of POLA and pre-installation choices by providing a channel for victims of bad addons to warn others who have not yet installed them. Ratings and user comments are a basic (if informal) tool. Curated galleries that promote a buggy addon can respond to complaints by removing it. Some systems include active revocation that allows the browser vendor to disable a particularly bad addon remotely, which is more or less effective depending upon the power of the addon (it may be too late to help victims of a malicious extension, but is helpful to get a vulnerable addon off the streets before it becomes widely exploited). Firefox has a blacklist that is checked daily, and Safari’s signed certificates presumeably allow Apple to disable addons similarly to the way they can remotely disable iPhone apps.



Different Approaches to Improve Security

	Firefox	Chrome	Safari	iPhone
pre-publication review	(yes)	(yes)	(yes)	yes
limited API	no	yes	yes	yes
user reviews	yes	yes	no?	yes
user involvement during install	no	yes	no	no (only geoloc)
revocation	yes	no?	yes?	yes

mozilla Labs
Jetpack

This table is a rough checklist of security features in web browsers, with similar features in iPhone applications thrown in for comparison, because its model is becoming increasingly relevant for addon platforms.

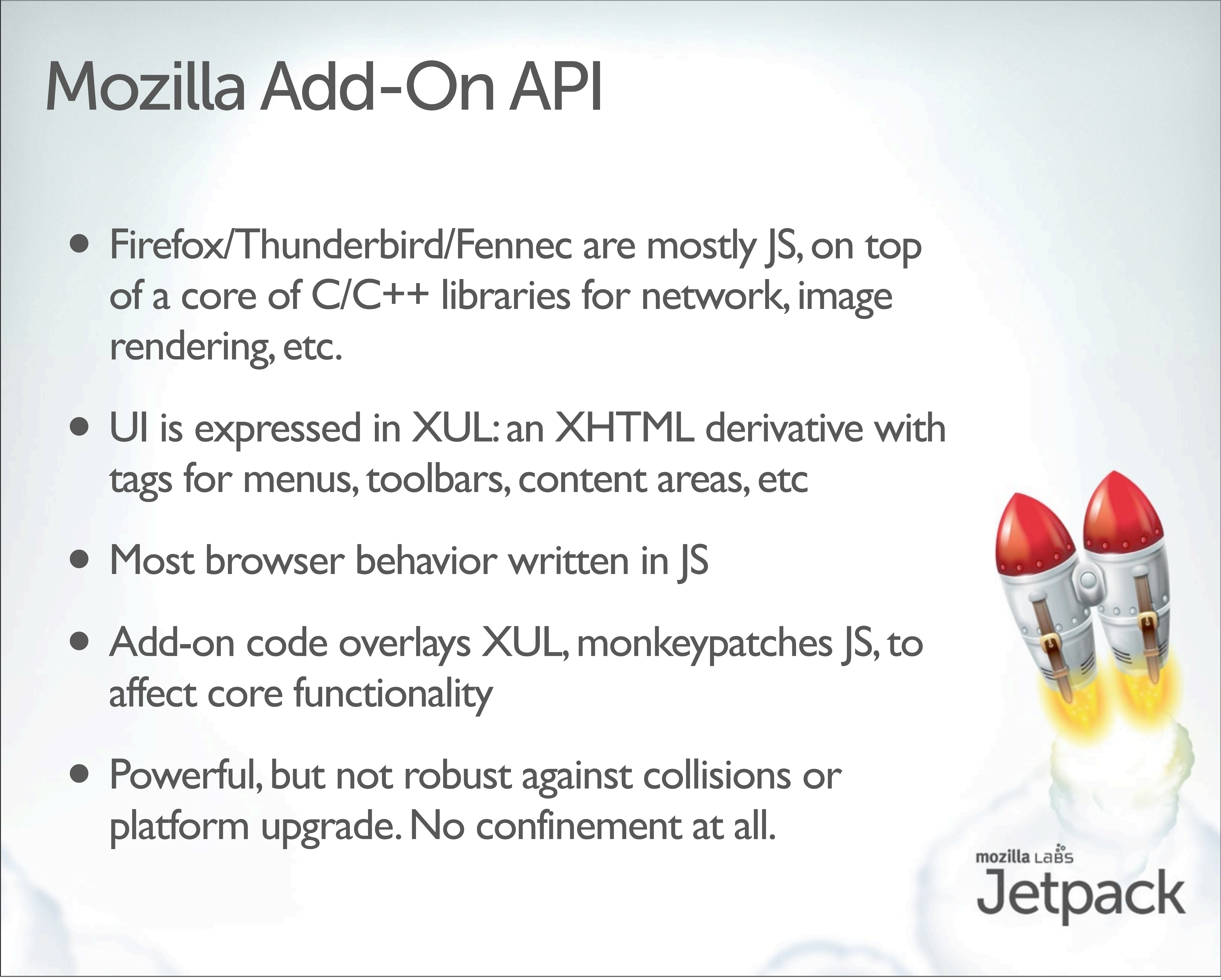
All the browsers add extra warning dialogs when pulling extensions from non-official sites, and presumeably they review the code that is promoted on the official site, so in a sense they are all performing pre-publication review (while the iPhone app store is very explicitly performing review, and short of jailbreaking, there is no way to bypass the process).

The new platforms are all using restricted APIs of some form, which limits the range of possible addons, but also prevents many of the simpler vulnerabilities. Only Chrome presents any information about API restriction to the user.

Mozilla Add-Ons and Jetpack

mozilla LABS Jetpack

So, now that we're done with the background, let's talk specifically about Firefox and Jetpack. We talk about "Mozilla Add-Ons" because there are actually several programs that share a common platform. The most well-known is the Firefox desktop browser, of course, but the Firefox Mobile browser (known as "Fennec") and the Thunderbird email client all share the same architecture, and there are add-ons which work on all three. Also, in the mozilla world, "plugins", "extensions", and "add-ons" are distinct things, of which we're specifically focussing on add-ons today. Each project adopts slightly different jargon.



Mozilla Add-On API

- Firefox/Thunderbird/Fennec are mostly JS, on top of a core of C/C++ libraries for network, image rendering, etc.
- UI is expressed in XUL: an XHTML derivative with tags for menus, toolbars, content areas, etc
- Most browser behavior written in JS
- Add-on code overlays XUL, monkeypatches JS, to affect core functionality
- Powerful, but not robust against collisions or platform upgrade. No confinement at all.

mozilla Labs
Jetpack

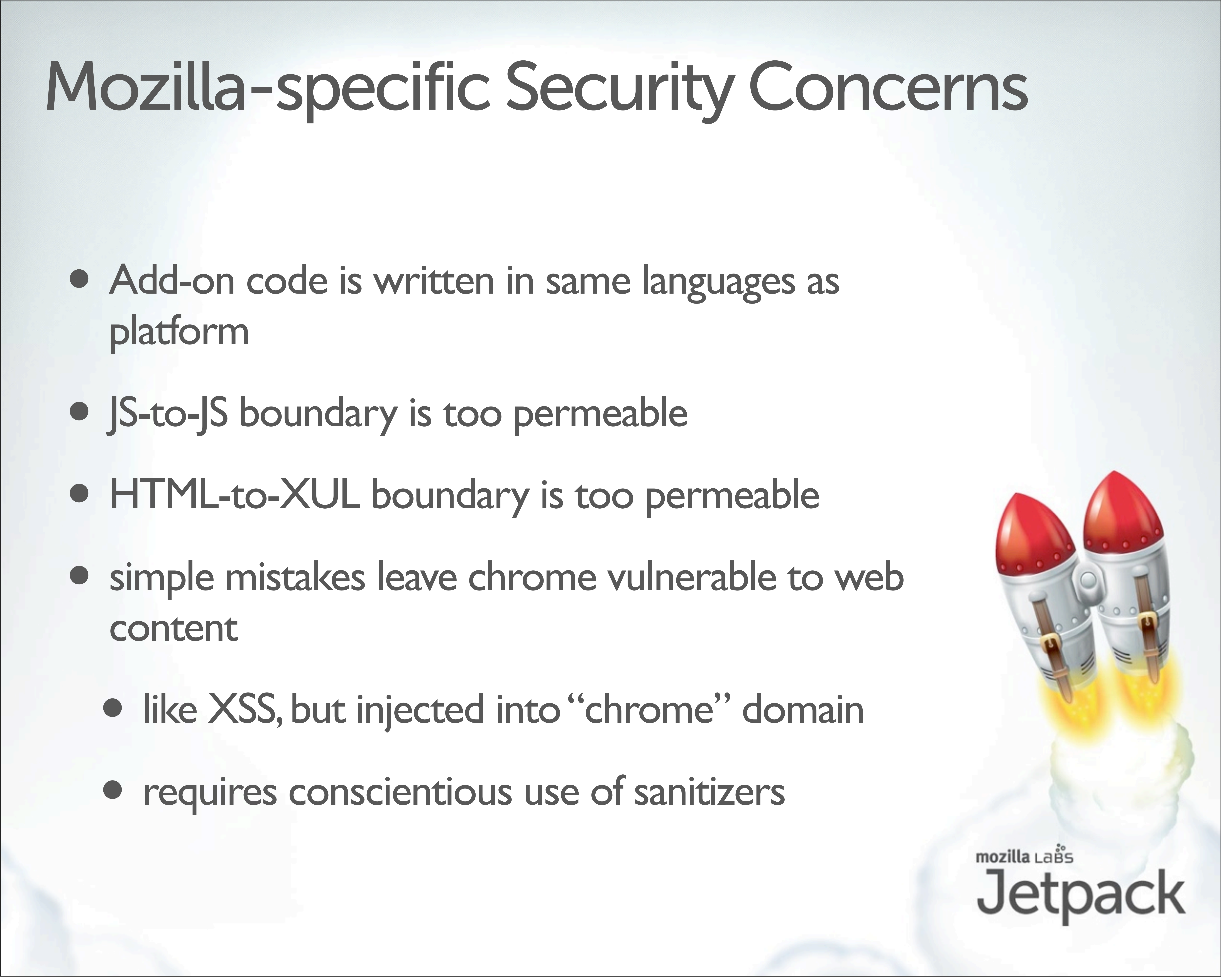
The Mozilla platform is fairly unique in being written mostly in Javascript. There is a core set of C++ libraries to do the “heavy lifting”, but most browser behavior is implemented in JS. This made cross-platform compatibility easier and sped up the development process. Using a high-level garbage-collected language also improved the memory-safety.

It also took the unusual step of writing all of the UI in an HTML-like language named XUL, in which menubars, dialog boxes, and toolbars (collectively known as the “chrome”) are expressed with markup tags. This also enabled rapid development, since UI engineers could rearrange the layout of the browser without a recompile or even knowledge of the code.

Add-ons took advantage of this implementation: they consist of Javascript and XUL, and are loaded into the browser just like the chrome. Some small accomodations were made for convenience, but for the most part add-ons hook into very low-level chrome objects and patch XUL to add UI elements.

The result is that Firefox addons are extremely powerful (since they can do everything that the normal browser code can do), but are not particularly robust against collisions between two addons trying to patch the same function, and are rather sensitive to internal changes that happen when the browser is updated. Many Firefox users report that they avoid upgrades (including security fixes, unfortunately) because of fear that their favorite addons will break.

Since add-ons are given access to all the facilities that browser chrome code can use, add-ons are not API limited. Code review is the only way to be confident that an add-on will behave according to its stated purpose.



Mozilla-specific Security Concerns

- Add-on code is written in same languages as platform
- JS-to-JS boundary is too permeable
- HTML-to-XUL boundary is too permeable
- simple mistakes leave chrome vulnerable to web content
  - like XSS, but injected into "chrome" domain
  - requires conscientious use of sanitizers

mozilla Labs
Jetpack

Firefox add-ons are written in the same language (JS/XUL) as the platform they run on, which makes them powerful enough to be dangerous. Add-ons suffer the same sort of script-injection concerns as regular web pages, except the victim is the whole browser. Simple addons which fetch weather data from a server and display it in a toolbar without removing script tags can accidentally give that server full control over the browser chrome, from which it can compromise the entire user account.

In general, crossing a language boundary imposes a barrier: some programmer must make a conscious effort to provide access to functionality across the boundary. When there is no language boundary, the opposite is true: it requires effort to *prevent* access to functionality.

There are a lot of tools and wrappers in the mozilla codebase to mitigate this problem, but it remains high on the list of things for add-on reviewers to watch for.

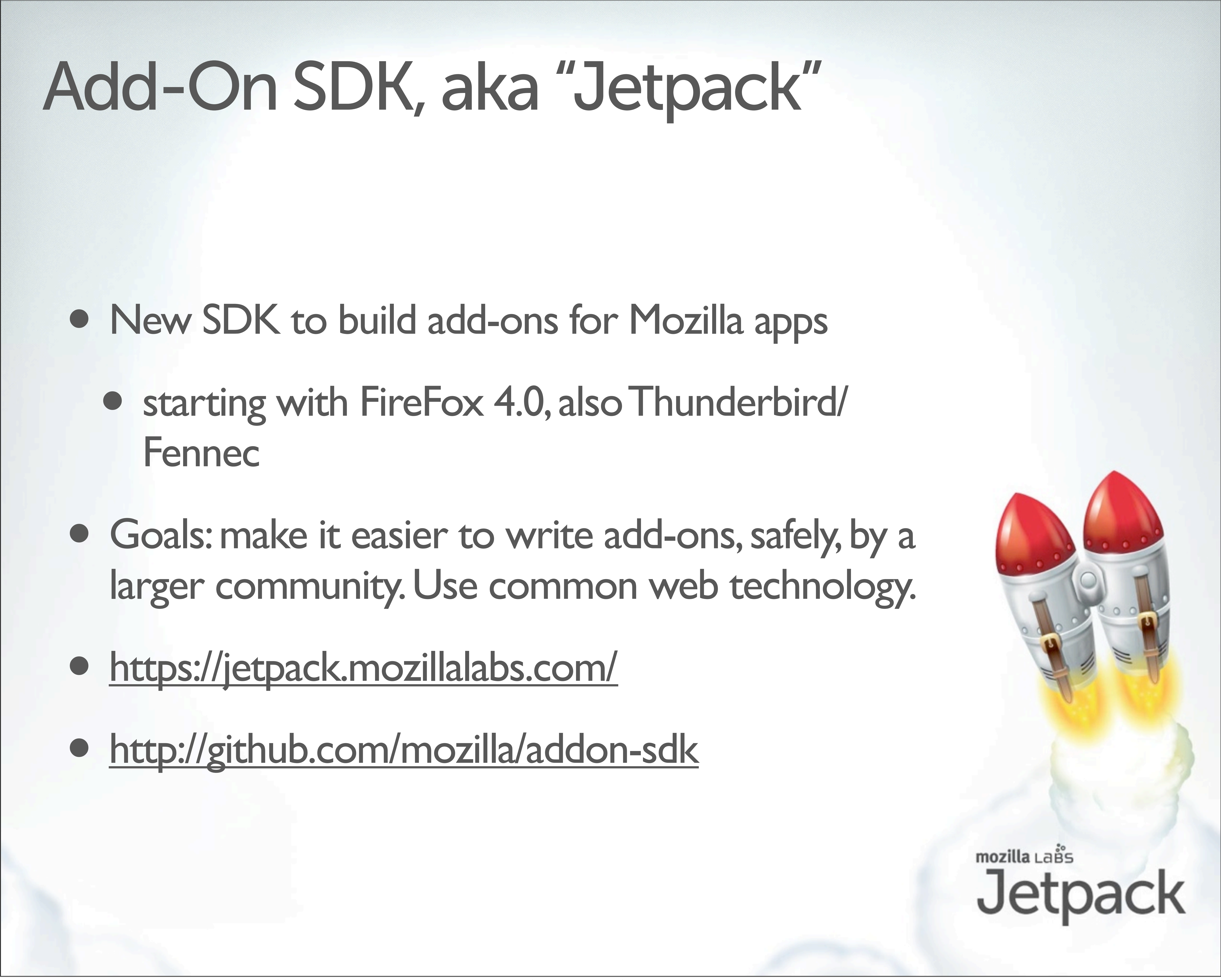
AMO Review Process

- addons.mozilla.org (“AMO”) hosts most add-ons
- a panel of reviewers inspects each add-on before initial publication or update
- security is enforced by SSL and maintaining control over the AMO gallery
- updates generally hit AMO too

mozilla LABS
Jetpack

Mozilla add-ons are generally hosted on addons.mozilla.org, AMO for short. Addons go through a review process before they are published to AMO, where experienced reviewers look for bugs and vulnerabilities. The browser only talks to AMO over https URLs. If you’re willing to believe in SSL and the CA world, and rule out the users who will bypass the install-from-other-sites warnings, then the AMO reviewers control which add-ons can and cannot be installed in Firefox.

Jetpack



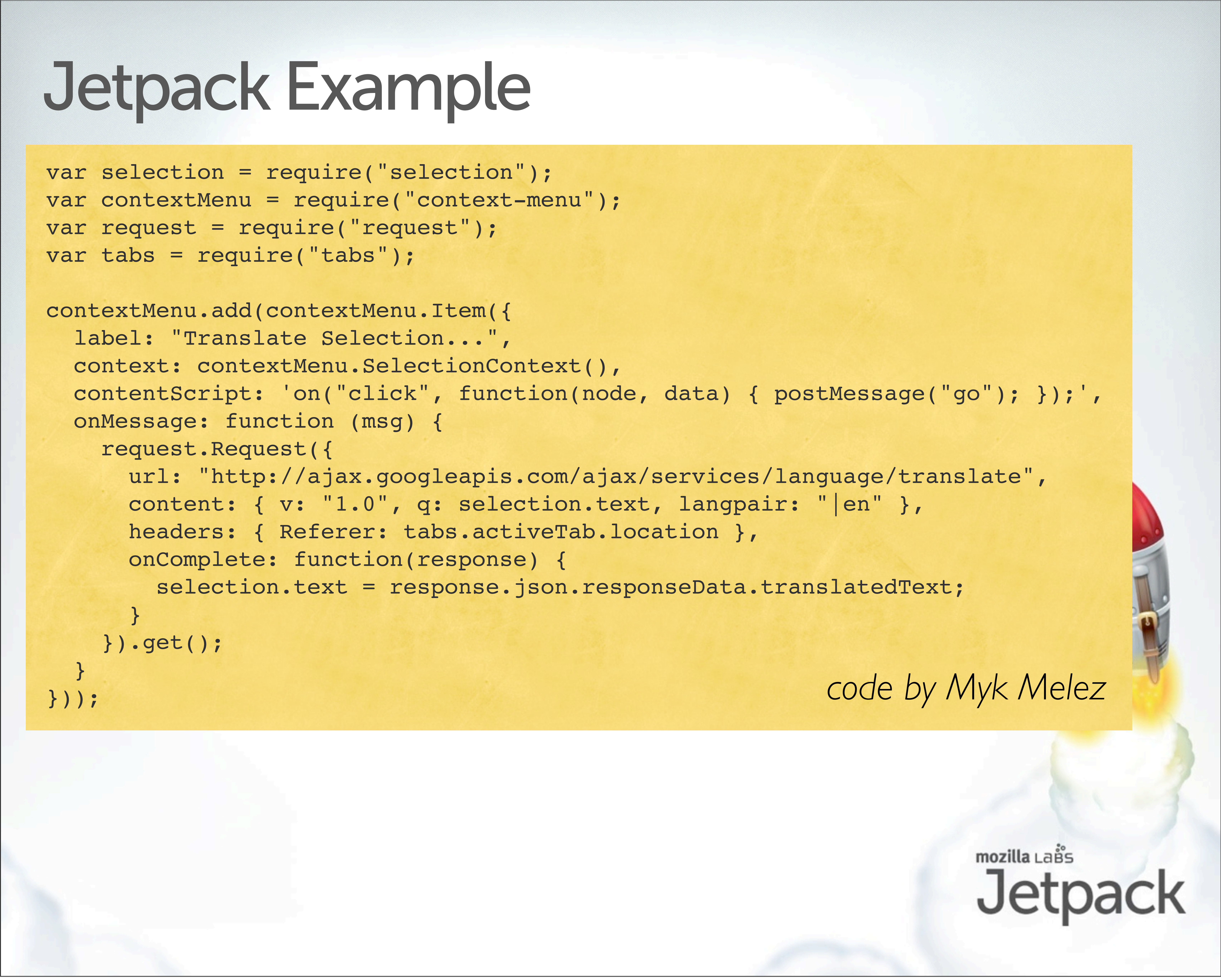
Add-On SDK, aka “Jetpack”

- New SDK to build add-ons for Mozilla apps
  - starting with FireFox 4.0, also Thunderbird/ Fennec
- Goals: make it easier to write add-ons, safely, by a larger community. Use common web technology.
- https://jetpack.mozillalabs.com/
- http://github.com/mozilla/addon-sdk

mozilla LABS
Jetpack

So, to fix some of this stuff, about a year and a half ago, before I joined Mozilla, the Jetpack project was started to develop a better add-on platform for Firefox and other Mozilla applications. The main goal has been to make add-on development easier, specifically for web developers, by abstracting away XUL and the messy internals, and leveraging common web technologies like JS, HTML, and CSS. A secondary goal has been to make addons safer: to get rid of the common security pitfalls in traditional addons. The Principle Of Least Authority has been a recurring theme.

We’re aiming to have Jetpack 1.0 released to match Firefox 4.0, in a couple of months. The code is up on github.



Jetpack Example

```
var selection = require("selection");
var contextMenu = require("context-menu");
var request = require("request");
var tabs = require("tabs");

contextMenu.add(contextMenu.Item({
  label: "Translate Selection...",
  context: contextMenu.SelectionContext(),
  contentScript: 'on("click", function(node, data) { postMessage("go"); });',
  onMessage: function (msg) {
    request.Request({
      url: "http://ajax.googleapis.com/ajax/services/language/translate",
      content: { v: "1.0", q: selection.text, langpair: "|en" },
      headers: { Referer: tabs.activeTab.location },
      onComplete: function(response) {
        selection.text = response.json.responseData.translatedText;
      }
    }).get();
  }
}));
```

code by Myk Melez

Here’s a quick sample of what a Jetpack-based addon looks like.

In Jetpack code, platform facilities are accessed by importing CommonJS modules like “selection” and “context-menu”. This add-on adds a context menu item which, when clicked, feeds the currently selected text through a translator and replaces it with the english equivalent. All of the UI and network functionality is provided by libraries which come with Jetpack.

demo?

Jetpack is an SDK

- Jetpack is an SDK
- FF platform is not being changed for Jetpack
- output is a traditional `.xpi` file, but input is easier
- includes modules with useful APIs
  - Jetpack developers have written the monkeypatches and overlays for you
  - module implementation will be updated to match platform changes, API will stay the same

mozilla Labs
Jetpack

It's important to note that Jetpack is an SDK, rather than being a new feature of the mozilla platform. Because of the size and inertia of the Mozilla codebase, and the time and stability constraints of the upcoming 4.0 release, Firefox itself is not undergoing significant Jetpack-specific changes. Instead, the Jetpack SDK contains packaging tools and a library of modules that provide useful APIs, each of which implements the messy details necessary to hook into the right internal functions. Addon developers write to the higher-level interfaces provided by these libraries. The SDK has a linker that combines some bootstrap code, a module loader, user code, and these libraries, assembling them together into a traditional XPI file, just like how a C compiler and linker do it. The XPI is then loaded by the browser in exactly the same way as an old-style manually-written addon.

Current APIs (0.9 release)

- clipboard
- context-menu
- localization
- notifications
- page-mod
- page-worker
- panel
- private-browsing
- request
- selection
- simple-storage
- tabs
- widget
- windows

mozilla LABS
Jetpack

We have a long list of APIs in the current release, and we're adding more all the time. Note that add-on developers are free to write modules that provide access to other APIs: these just represent the basic set of functionality that we decided to write first. Each API is provided by a distinct module, so it's easy to share new functionality with others.

Add-on Builder, aka “FlightDeck”

- Web site / IDE to write Jetpack add-ons
- code editor with completion, docs lookup
- share libraries with other developers
- https://flightdeck.mozillalabs.com/

We’ve also built a web-based IDE for writing Jetpack code. It has a nice integrated code editor, module search tools, facilities for sharing code with other developers, and easy buttons to submit your finished addon to the official gallery.

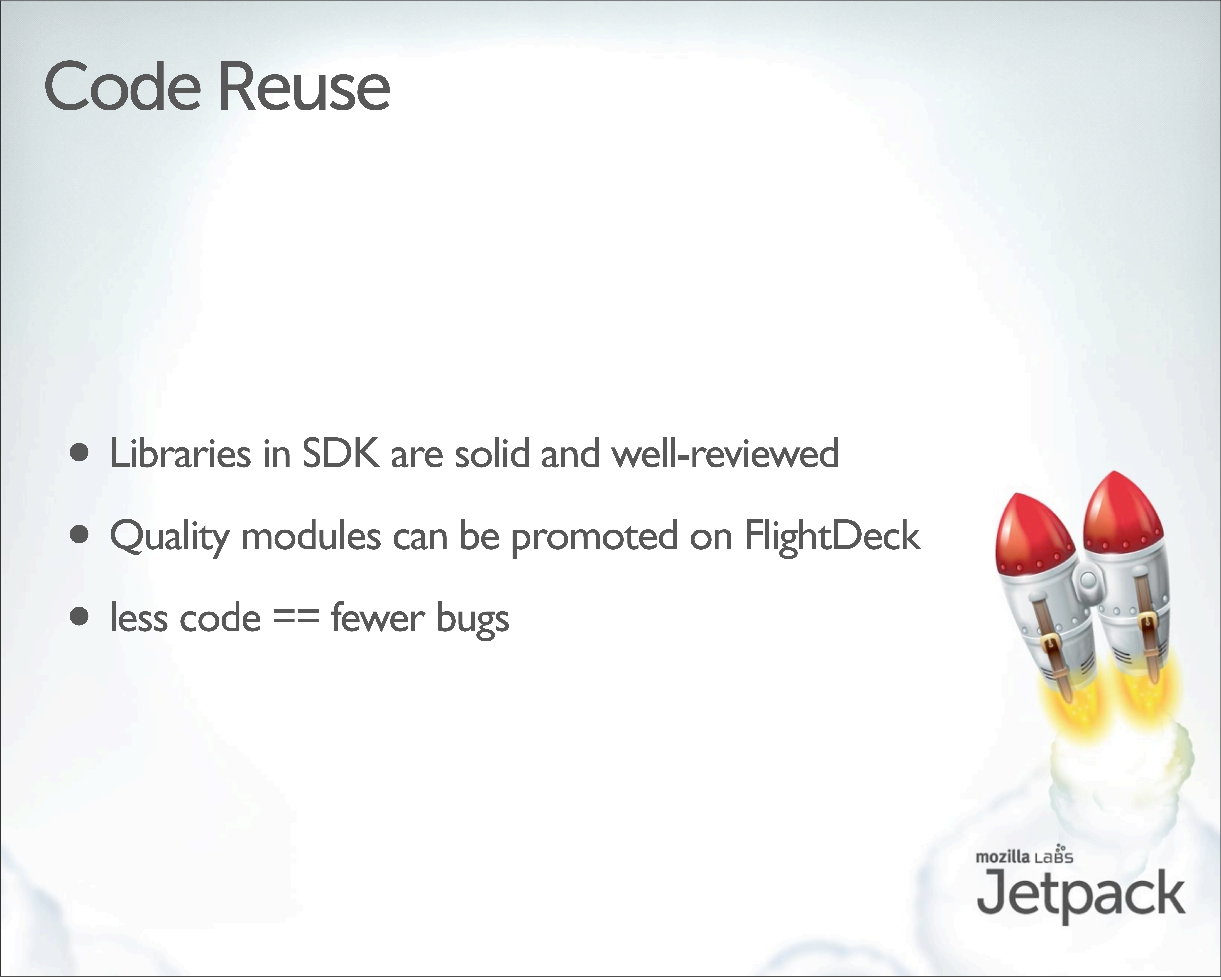
Jetpack Security

Jetpack Security

- code reuse
- content script isolation
- module isolation
- module-graph manifest
- code review tools
- Lots of work to do

mozilla LABS Jetpack

There are a lot of ways in which Jetpack has the potential to improve the security of addon code. I'll be talking about what we've done so far, and what some of our future plans are.



Code Reuse

- Libraries in SDK are solid and well-reviewed
- Quality modules can be promoted on FlightDeck
- less code == fewer bugs

mozilla LABS
Jetpack

First off, having an SDK at all means that a lot of basic libraries will be written by experienced platform developers and thoroughly reviewed, so that add-on authors don't have to reinvent the wheel (and reinvent its bugs) each time. This is a big win. Third-party modules that are well-written and get positive reviews will get exposure on FlightDeck, guiding developers to use existing ones instead of starting from scratch. The less code a developer has to write, the fewer bugs they'll be creating.

Content-Script Isolation

- Scripts added to web pages are isolated from rest of add-on
- must communicate with it via `postMessage()`
- avoids leaking add-on authority to web content
- protects both sides from exceptions, reentrancy

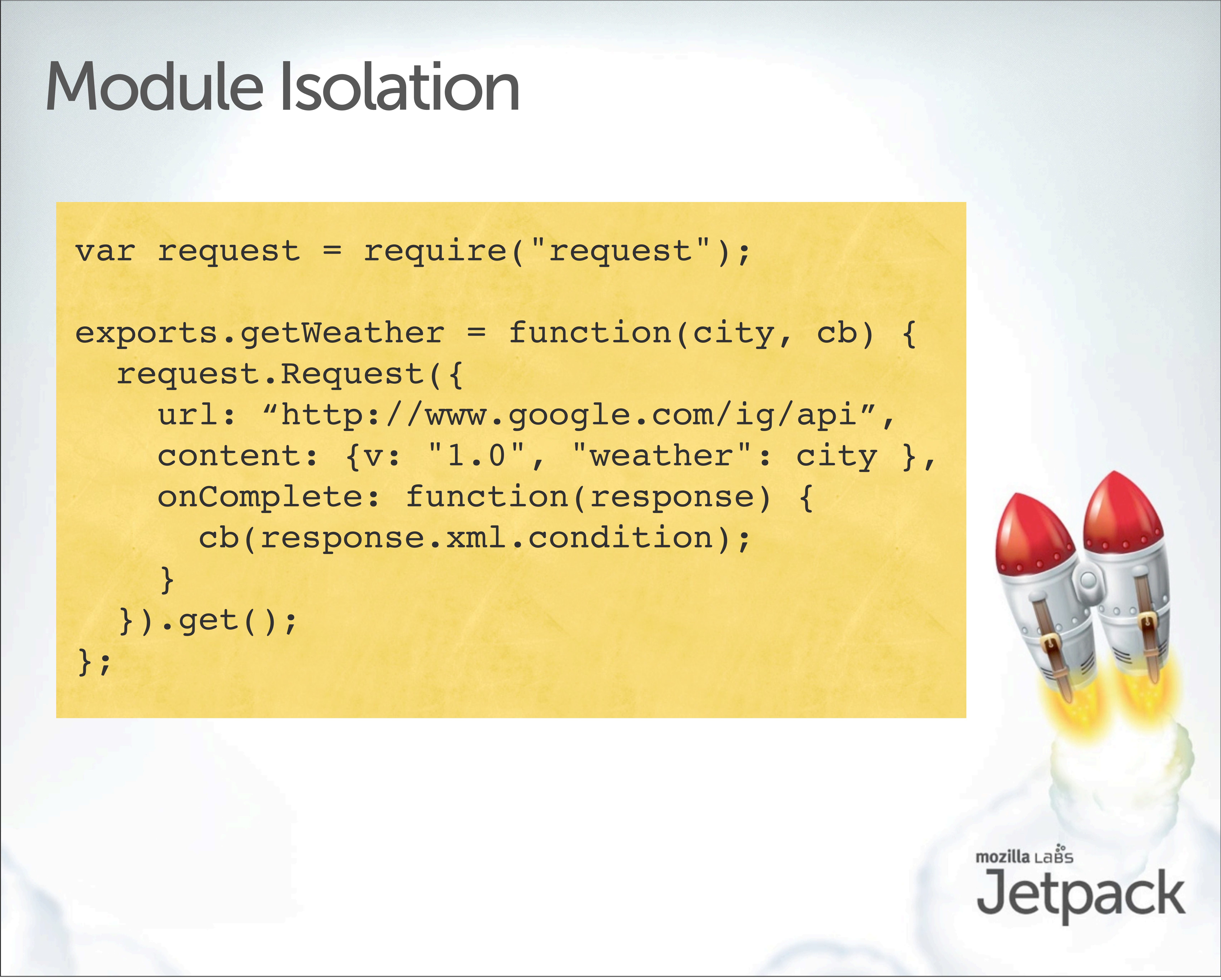
mozilla LABS

Jetpack

Like Chrome and Safari, when a Jetpack add-on modifies a web page by injecting a script, that script does not get direct access back to the add-on. It is separated by a message-passing barrier, so the developer must provide explicit handlers to grant power to the script.

It's a good policy to assume that the injected script will get compromised sooner or later by unexpected code in the web page. When this happens, the message-passing barrier makes it less likely that the page will get access to all of the add-on's authority.

This actually comes as a side-effect of the "Electrolysis" project, which moves content rendering out of the main browser process and into a separate one, mostly to make the UI more responsive. Each Jetpack addon will run in its own process, so the message-passing API was required anyway.



Module Isolation

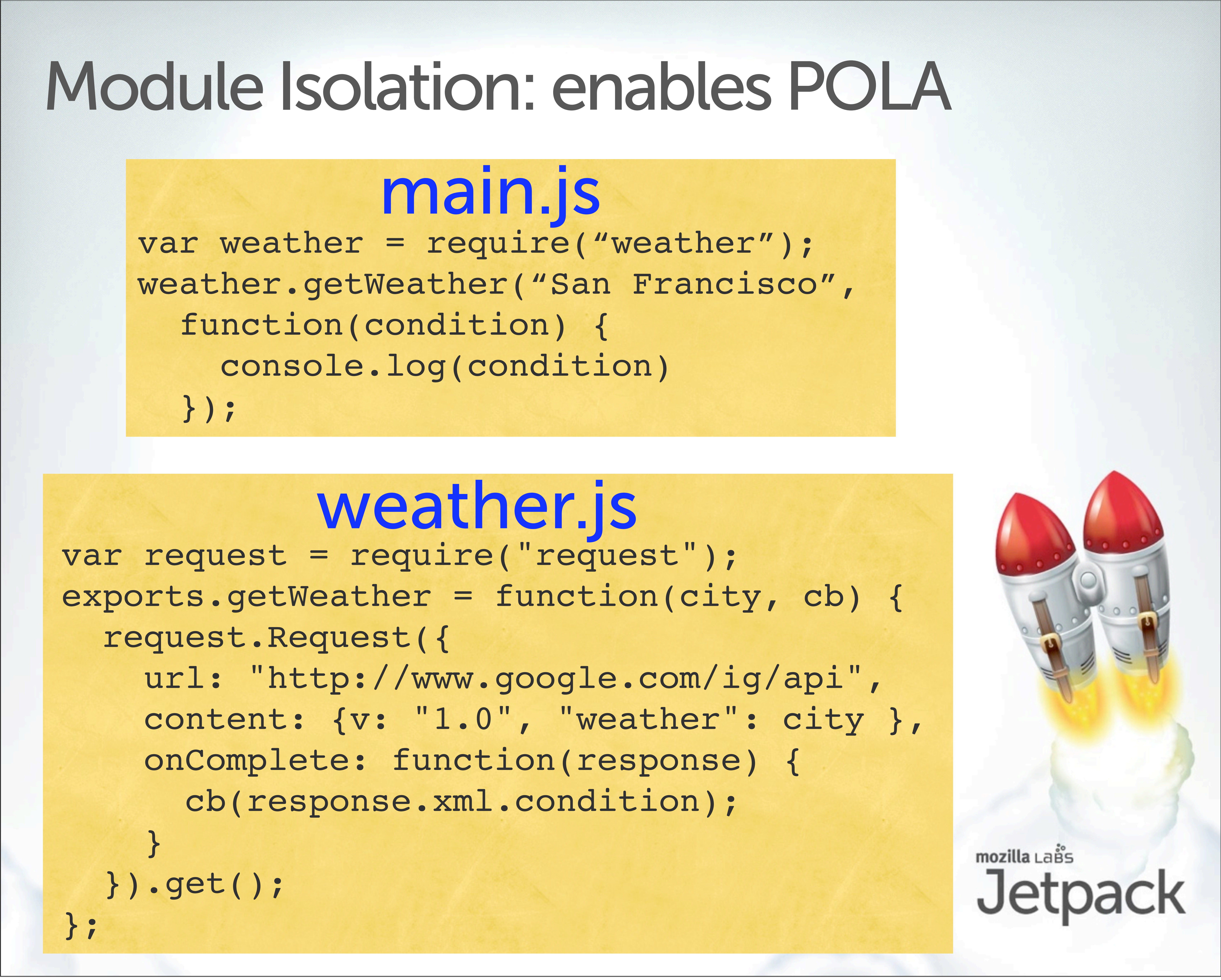
```
var request = require("request");

exports.getWeather = function(city, cb) {
  request.Request({
    url: “http://www.google.com/ig/api”,
    content: {v: "1.0", "weather": city },
    onComplete: function(response) {
      cb(response.xml.condition);
    }
  }).get();
};
```

mozilla Labs
Jetpack

By using CommonJS modules, Jetpack is establishing a framework, an expectation, in which modules are isolated from each other. This simple module uses the “request” API, which is a nicer interface to the usual XMLHTTPRequest function, to access a server and retrieve a specific piece of data. It exports a single “getWeather” name.

Rather than the usual web-page Javascript environment in which all scripts are mushed together into the same global namespace, CommonJS modules have clearly-defined entry and exit points: each JS file is evaluated in a separate context that contains only the “require” and “exports” symbols, giving it a way to access external functionality like the “request” module, and to provide functions to some other module that is requiring this one. Other names defined inside this file, in particular the “request” object, will not be visible to outsiders.
The goal is to give each module strong control over how its internal state is influenced by its callers.

We don’t yet completely enforce this isolation: there’s probably still some introspection feature that would allow a caller of this getWeather() function to see inside its private state. But the rule is that you should not be doing that, and AMO reviewers are free to reject any code appears to violate this isolation. Over time we’ll be improving the isolation, until it’s impossible to violate it, and we won’t need to rely on code review.



Module Isolation: enables POLA

main.js

```
var weather = require(“weather”);
weather.getWeather(“San Francisco”,
  function(condition) {
    console.log(condition)
  });
```

weather.js

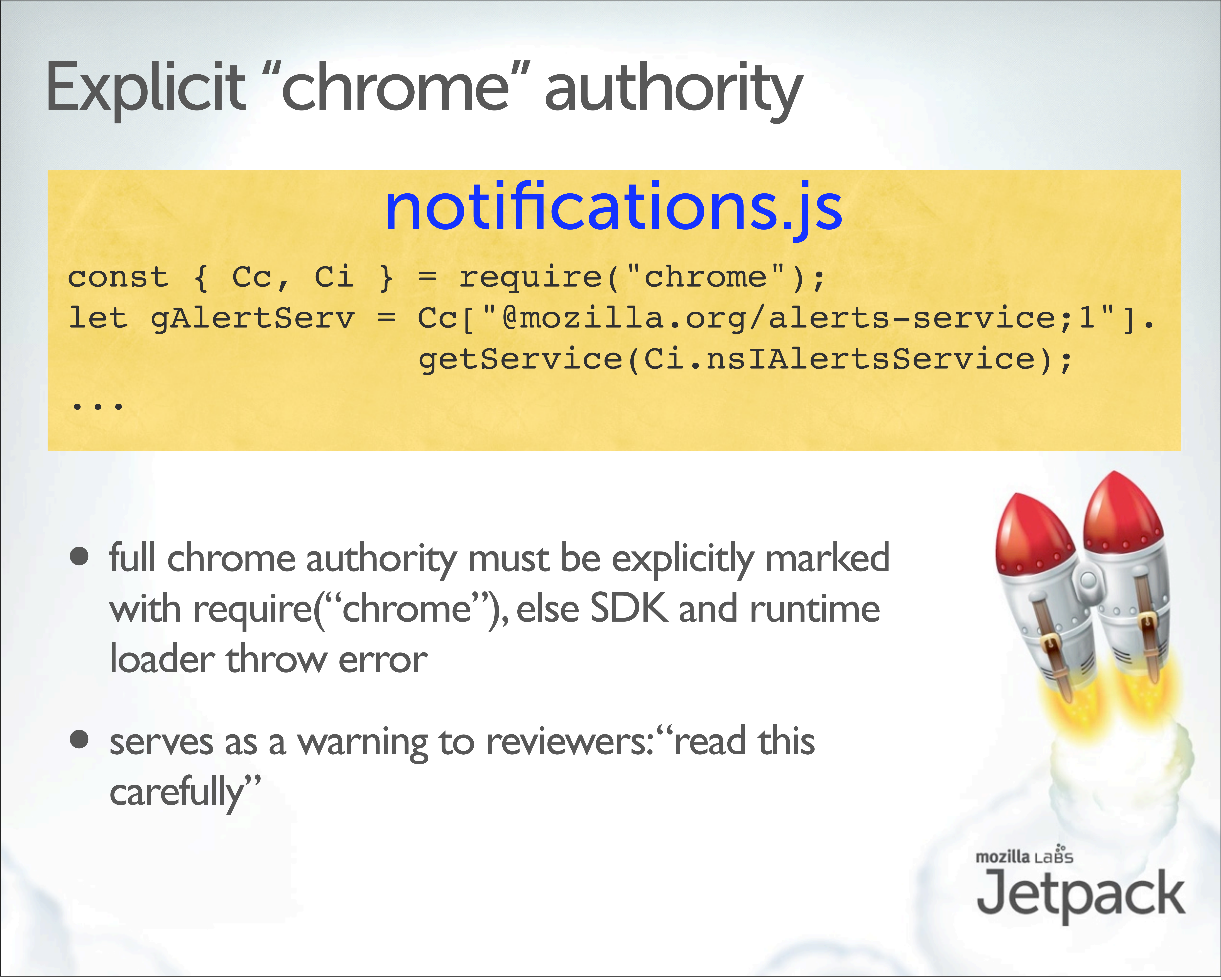
```
var request = require("request");
exports.getWeather = function(city, cb) {
  request.Request({
    url: "http://www.google.com/ig/api",
    content: {v: "1.0", "weather": city },
    onComplete: function(response) {
      cb(response.xml.condition);
    }
  }).get();
};
```

mozilla LABS
Jetpack

Module isolation is interesting because it lets us attenuate authorities and give each module just the power it needs, and no more. In this example, the weather.js module imports the powerful “request” module, which allows network access to every server on the internet. However it only ever uses that power to access a single weather server, and gives its caller very little control over the URL that is fetched. As a result, the main.js module, which only uses “weather” and not “request”, can do far less damage should something go wrong, like a bug, compromise, or outright malice.

If we didn’t have this isolation, then any bug or compromise in main.js would expose the full power of the “request” module. If weather.js were importing something more powerful, like access to user passwords or the local filesystem, then a bug in main.js would be security-critical. But with this isolation, it is merely a harmless nuisance.

What’s really going on here is that the author of main.js is committing to the power that they want up front, by writing those “require” statements. They happen to serve double duty as static declarations of intent. When the Jetpack SDK builds an XPI, it greps through each module for these require statements and writes then down in a manifest. This manifest is then available to reviewers, to understand which modules want to do what, so they can focus their review time on the important parts. The manifest is also enforced at runtime, so modules cannot dynamically import code which they didn’t ask for ahead of time.



Explicit “chrome” authority

notifications.js

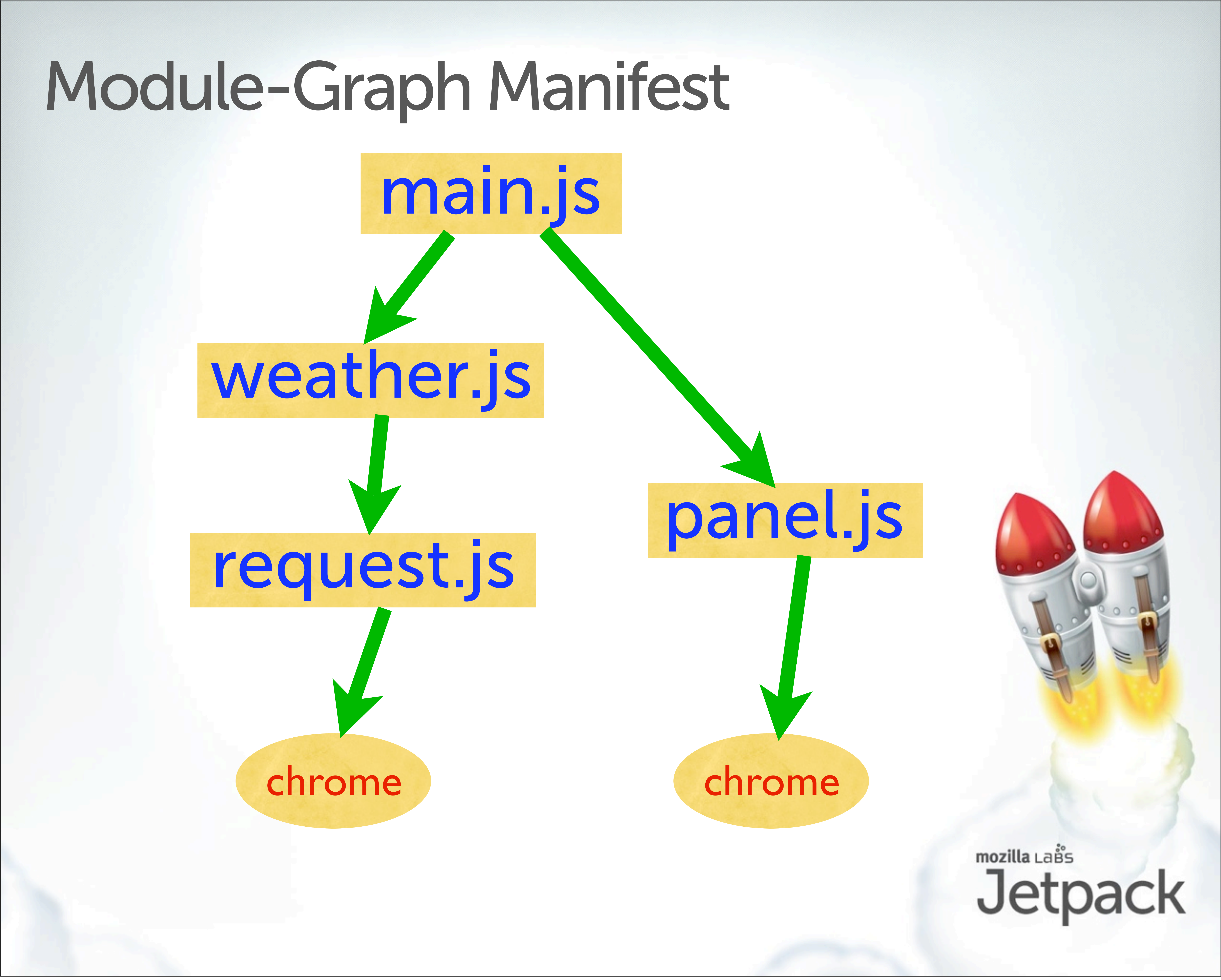
```
const { Cc, Ci } = require("chrome");
let gAlertServ = Cc["@mozilla.org/alerts-service;1"].
                 getService(Ci.nsIAlertsService);
...
```

- full chrome authority must be explicitly marked with require(“chrome”), else SDK and runtime loader throw error
- serves as a warning to reviewers: “read this carefully”

There’s a special form of the require statement called “require(chrome)”, which is used to access the browser’s all-powerful low-level component system, which is like root for the browser.

This snippet is probably hard to read unless you’re familiar with the Mozilla XPCOM system and spidermonkey’s “destructuring assignment” syntax, but basically this Javascript module is getting access to a C++ alert service. The “Cc” stands for Components.Classes, and can be used to access every low-level library in the system, allowing completely arbitrary control over the browser and the rest of the user’s account. Such unconfined power should not be used lightly, so only the bottom-most modules, which must be implemented by accessing the internals of the Mozilla platform, will have this require(chrome) call. The SDK and runtime loader try to enforce the rule that chrome authority cannot be used by modules that do not declare themselves with “require(chrome)”. Again, enforcement is not complete yet, but it’s getting stronger all the time.

This declaration also serves as a warning to reviewers, who should read such modules with extra care. A bug in a chrome-authority module, just like a bug in OS kernel code, has more severe consequences than a bug in a less-powerful module. We scan for require(chrome) and mark it down in the manifest, where it’s available to both reviewers and the runtime loader.



Module-Graph Manifest

main.js

weather.js

request.js

panel.js

chrome

chrome

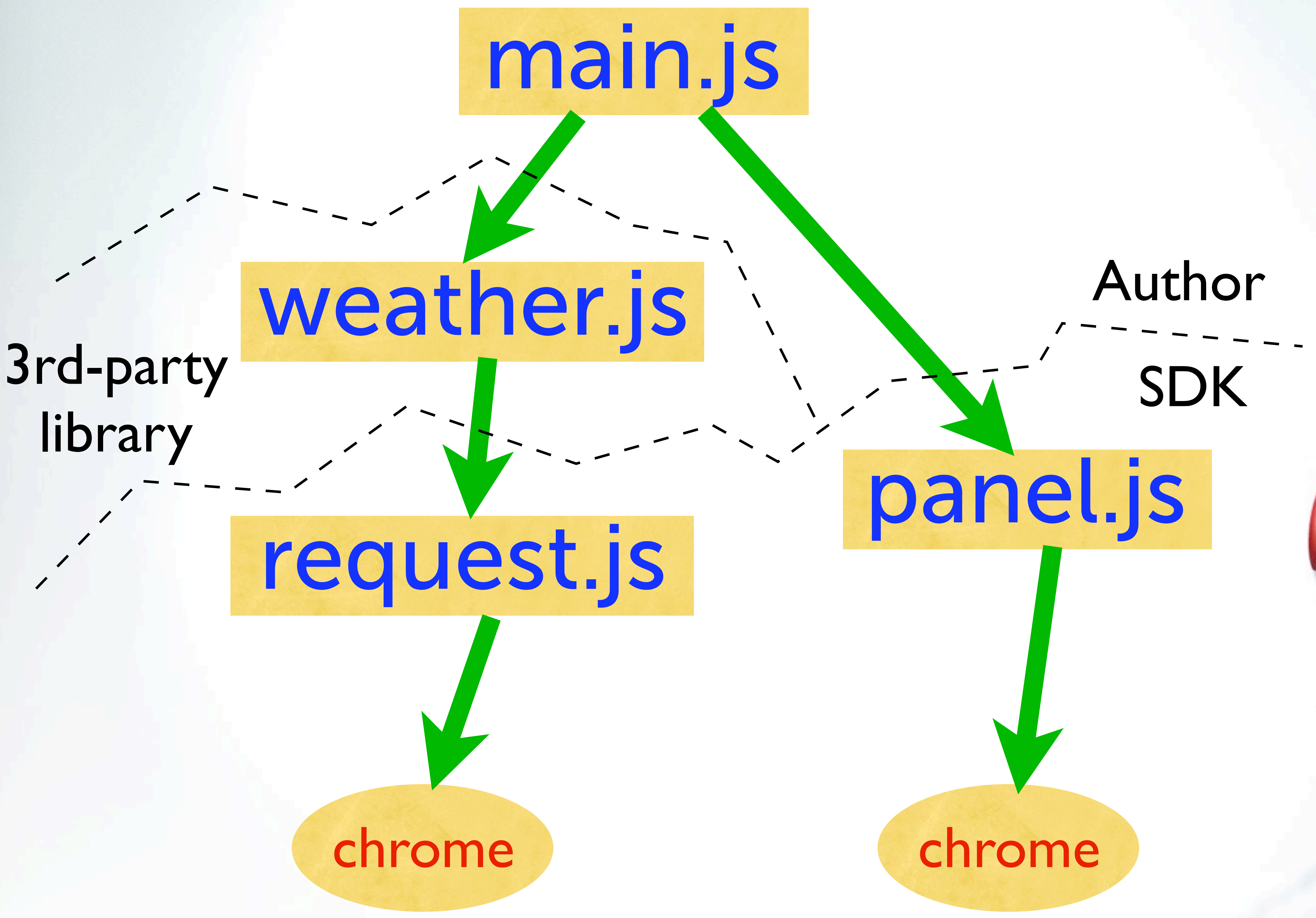
mozilla Labs Jetpack

The modules that make up a Jetpack add-on fit together into a directed graph, generally a tree, in which the top-level driver module contains “main”, and the bottom-most modules use chrome authority to grant access to platform facilities. Since the only way to get power is by importing modules with the “require()” function, we can build the graph by scanning for all the require() calls. The SDK performs a simple static analysis and builds a manifest with this graph, and the runtime loader has code to prevent calls to require() that were not found during this scan. As a result, the manifest describes the distribution of authority in the add-on. Basically there’s lots of power at the bottom, and it gets attentuated as you go up.

Earlier, I talked about how useful it’d be to establish an upper bound on the power of a given addon. Here, we have an upper bound on the power of an individual module: the “weather” module can do no more than what the “request” module gives it, and by examining its code, we can see that the weather module provides even less power to main.js . If the panel module is well-designed and only provides the ability to put images and text in a UI area, then the authority available to main.js is quite limited.

That means that, even if you didn’t look at the code in main.js at all, the worst it could do would be to spam a server with weather requests, and put rude messages into the UI. That’s a much better situation to be in than having the worst case be data theft and hard drive erasure.

TODO: Module Graph Review Tools



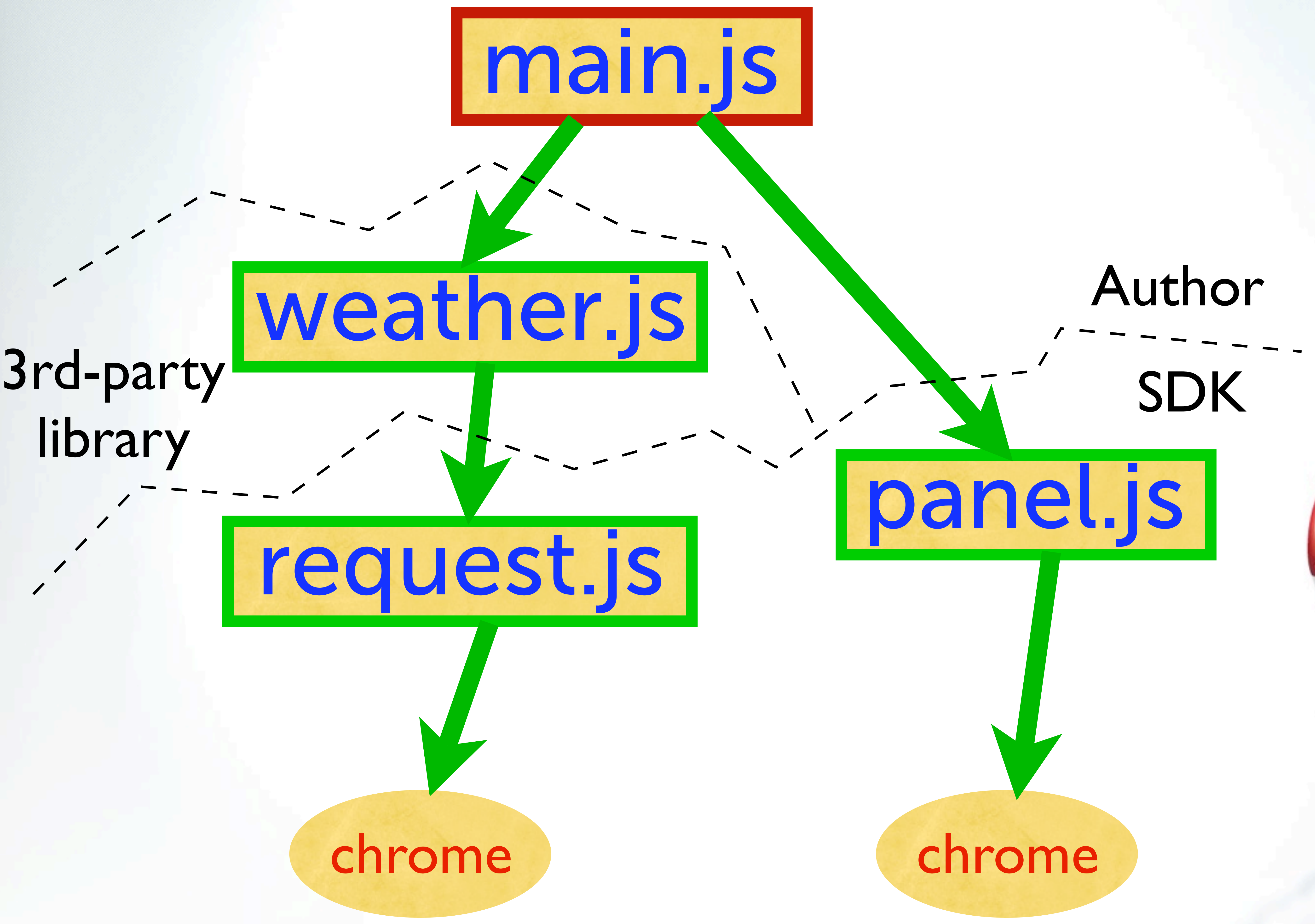
mozilla Labs Jetpack

Ok, now here’s where we depart from what we’ve done so far, and get into what we’re thinking about building next.

In any given addon XPI bundle, there will be some modules that came from the SDK, some that were written by third parties, and some that were written by the final add-on author. When the XPI is uploaded to AMO, we can take it apart, look at each module, and compare the contents of their javascript and documentation files to a database of known modules, to tell whether we’ve seen the module before, or if it’s brand new.

My plan is to build a web service to which you can upload an XPI, dissects it into the individual modules, then provides a page for each add-on and a page for each module inside it. On these pages, reviewers (and anyone else) can add comments, discuss bugs or potential problems with different parts of the code, and apply all the neat code-review tools that are showing up on Github and in various google projects.

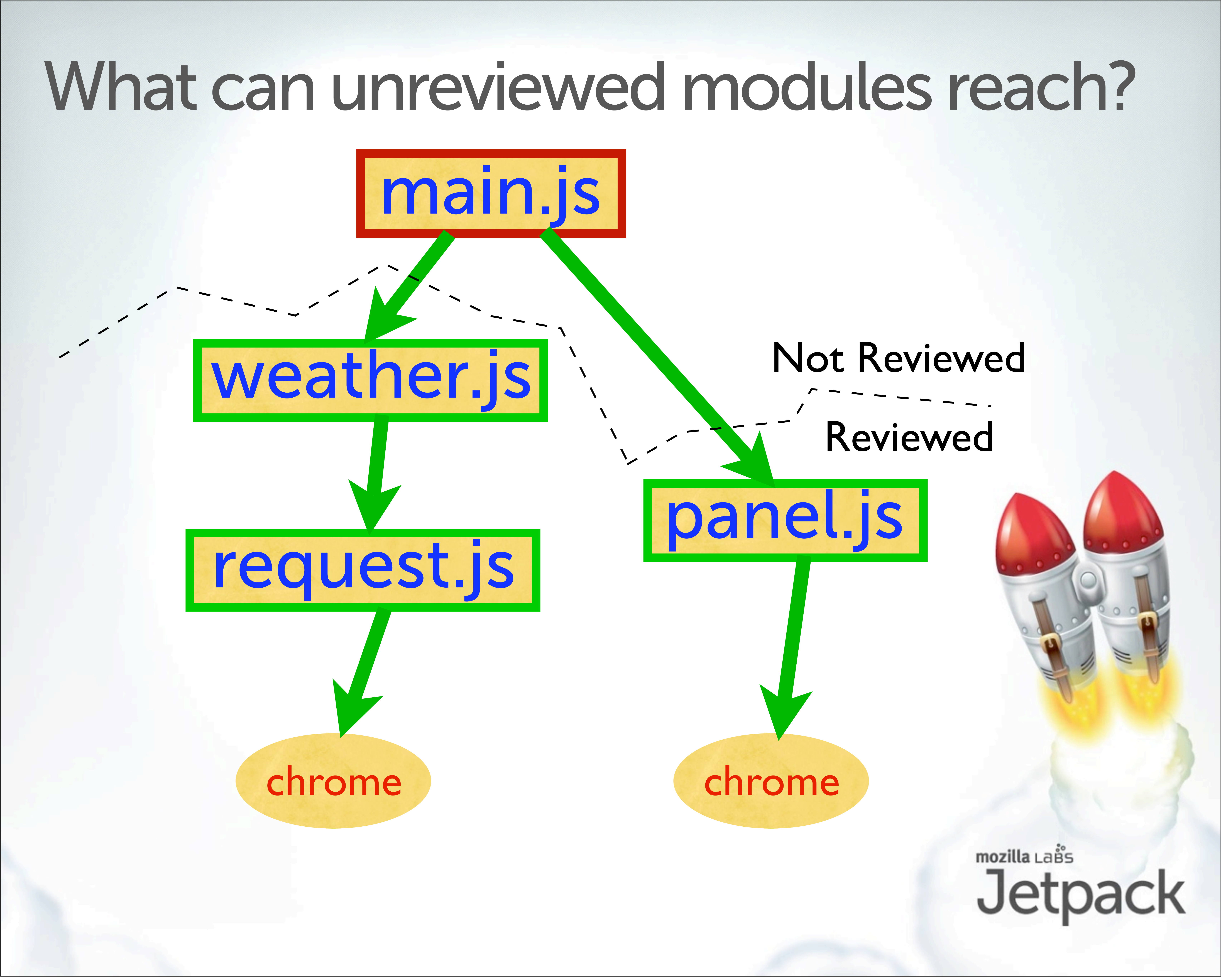
TODO: Module Graph Review Tools



When a reviewer is satisfied with a given module, there would be a checkbox to sign off on it.
That reviewer is making the claim that this module, say the panel.js module, provides a reliable implementation of some particular interface, as defined by the documentation that accompanies the module. This interface documentation is what the reviewer of the next layer up will rely upon. When somebody reviews main.js and sees that it imports panel.js, they won't need to look at the actual implementation of panel.js, they can just look at the description of what it does, because the implementation has been checked by somebody else.

My plan is to have these signoffs or approvals expressed as cryptographic signatures over the contents of the module, which the browser (or the jetpack loader framework) could verify. But end-to-end verification of module contents is a fairly low priority, so we may not get that far. At the very least, the AMO gallery could refuse to host addons which contain unreviewed components.

All the modules that come from the SDK will have been reviewed already, and popular 3rd-party libraries are also likely to have passed through a review process earlier, leaving just the top-level module left (shown here in red). If we draw a line between the modules that have been reviewed, and the modules that have not..



What can unreviewed modules reach?

main.js

weather.js

request.js

panel.js

chrome

chrome

Not Reviewed

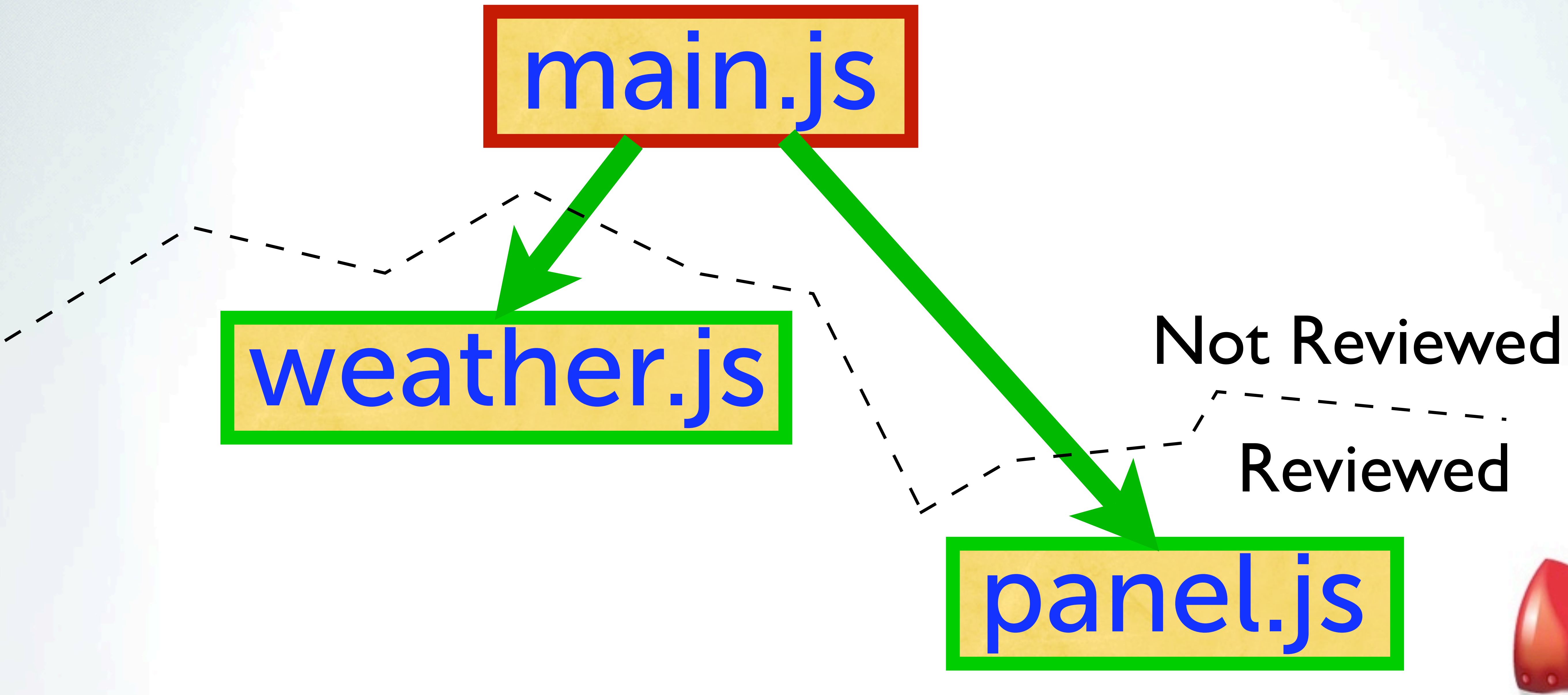
Reviewed

mozilla Labs Jetpack

.. we can see that the unreviewed code gets two authorities: the ability to fetch weather data, and the ability to fill a UI panel. All of the internal details of the reviewed modules can be ignored, as they were examined and approved earlier by the previous reviewers of those modules.

Now this is something you might be able to present to the user. There's some code that we know about, and some code that we don't, and we're proposing to allow the unknown code have access to the known code, and the user needs to decide whether the benefit of running that code is worth the risk. The risk is bounded by the power granted by the known code. So instead of asking the user to compare a claimed benefit against a possibly-infinite cost, they can compare it against a known finite cost.

What can unreviewed modules reach?



main.js

weather.js

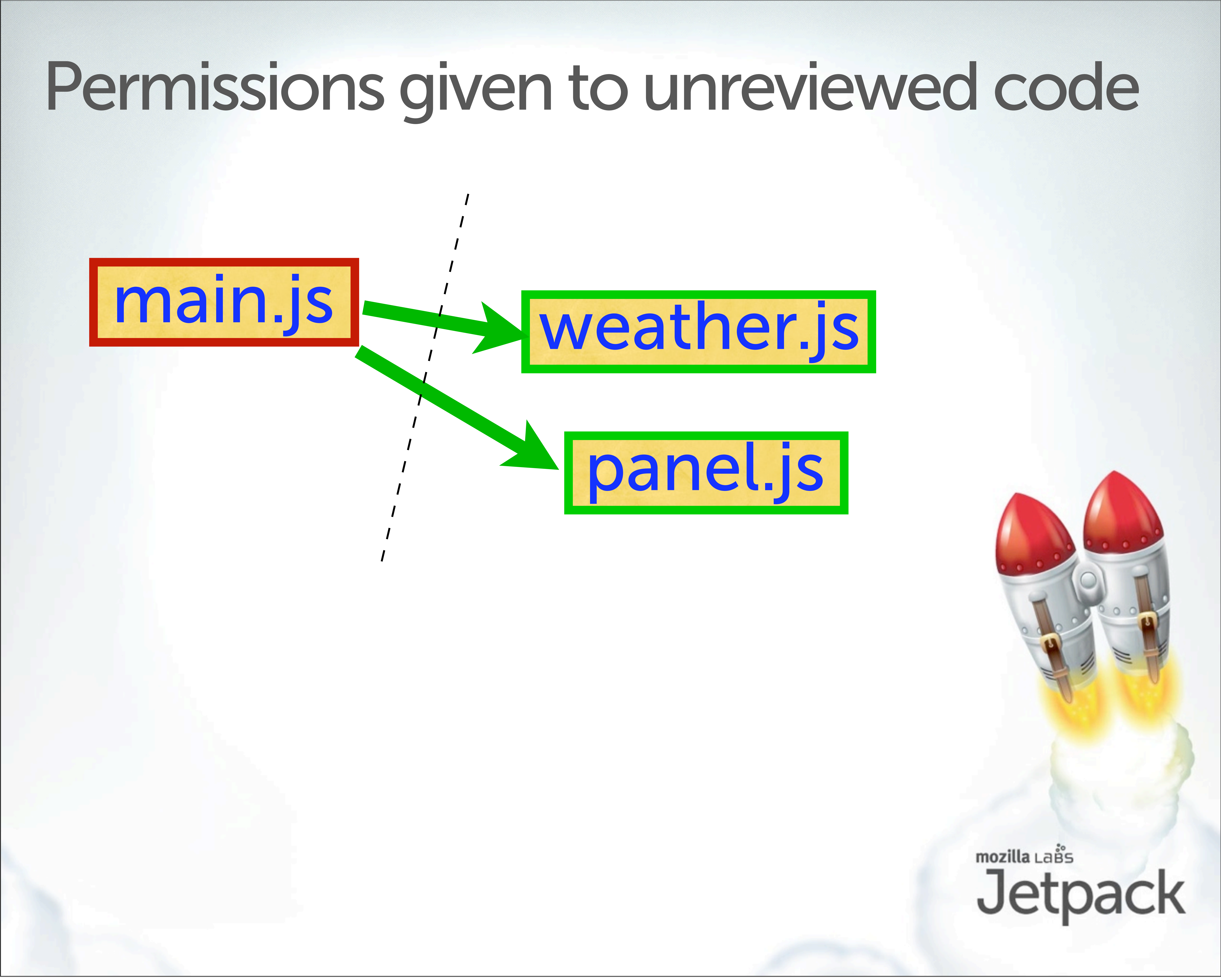
panel.js

Not Reviewed

Reviewed

.. we can see that the unreviewed code gets two authorities: the ability to fetch weather data, and the ability to fill a UI panel. All of the internal details of the reviewed modules can be ignored, as they were examined and approved earlier by the previous reviewers of those modules.

Now this is something you might be able to present to the user. There's some code that we know about, and some code that we don't, and we're proposing to allow the unknown code have access to the known code, and the user needs to decide whether the benefit of running that code is worth the risk. The risk is bounded by the power granted by the known code. So instead of asking the user to compare a claimed benefit against a possibly-infinite cost, they can compare it against a known finite cost.



Permissions given to unreviewed code

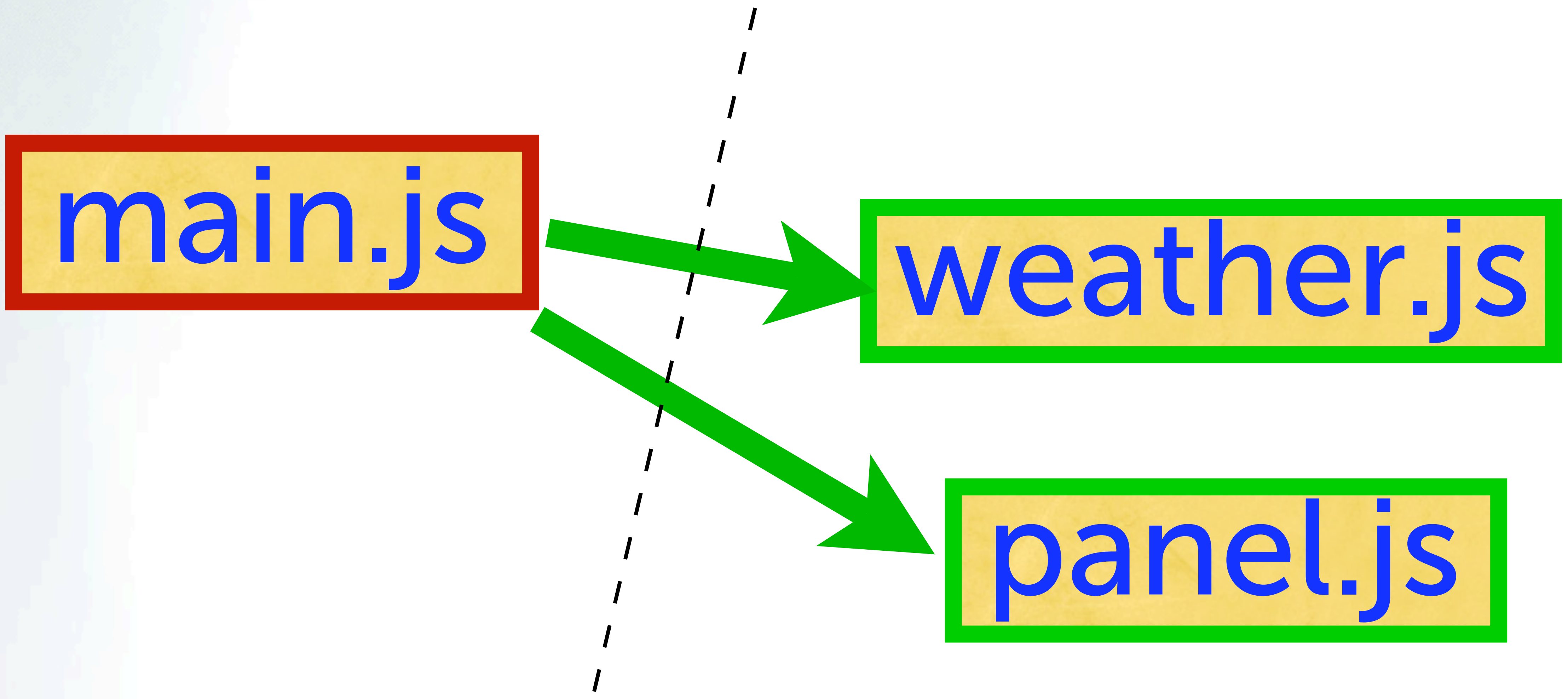
main.js

weather.js

panel.js

In fact, I think this is a superset of the permissions model provided by Chrome and Android. If instead of taking arbitrary modules, you picked a specific set of APIs and baked them into the platform, then made the unreviewed top-level code statically declare which ones it wanted to get, then you gave the user a brief description of those APIs before installation..

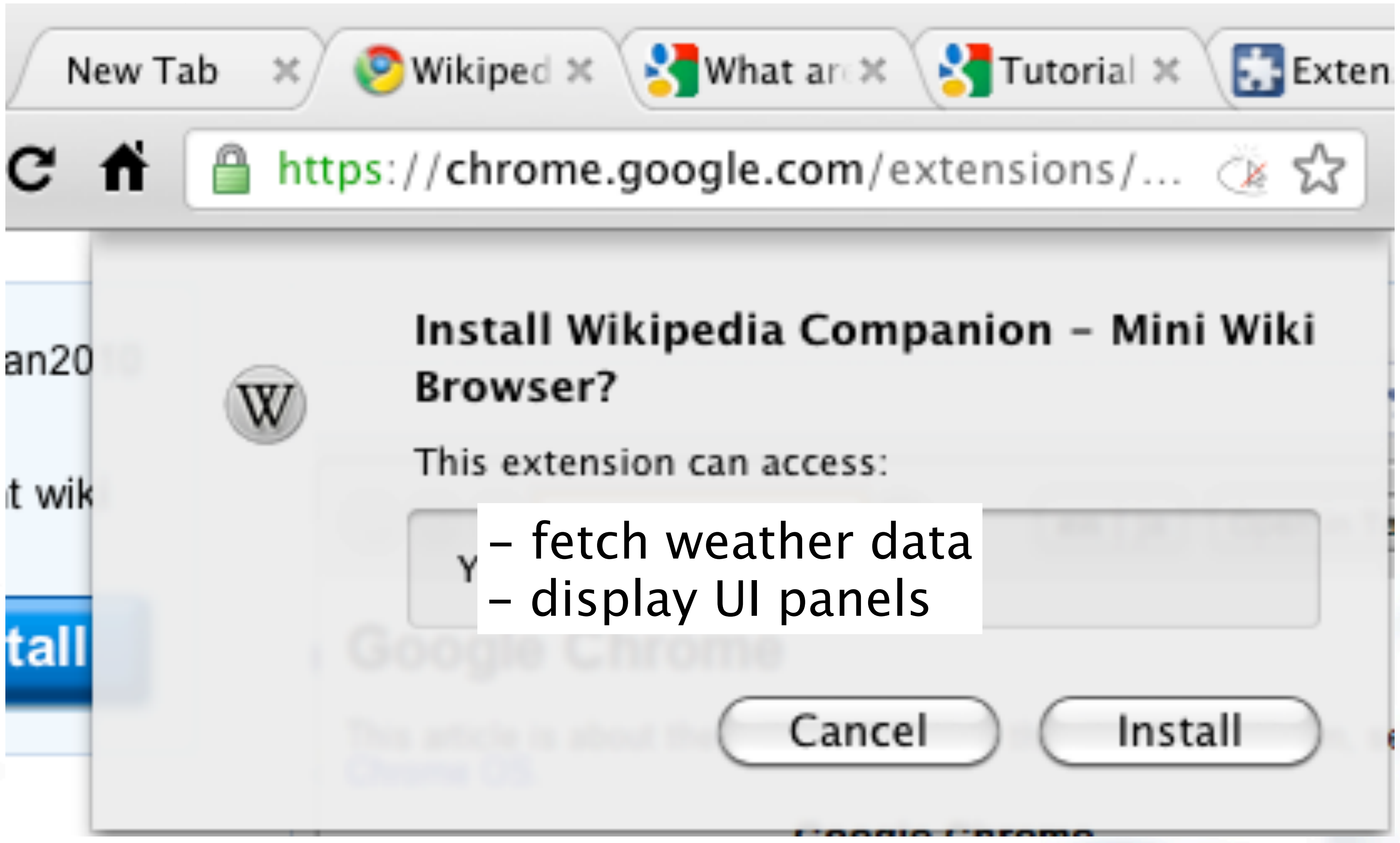
Permissions given to unreviewed code



main.js

weather.js

panel.js



https://chrome.google.com/extensions/...

Install Wikipedia Companion – Mini Wiki Browser?

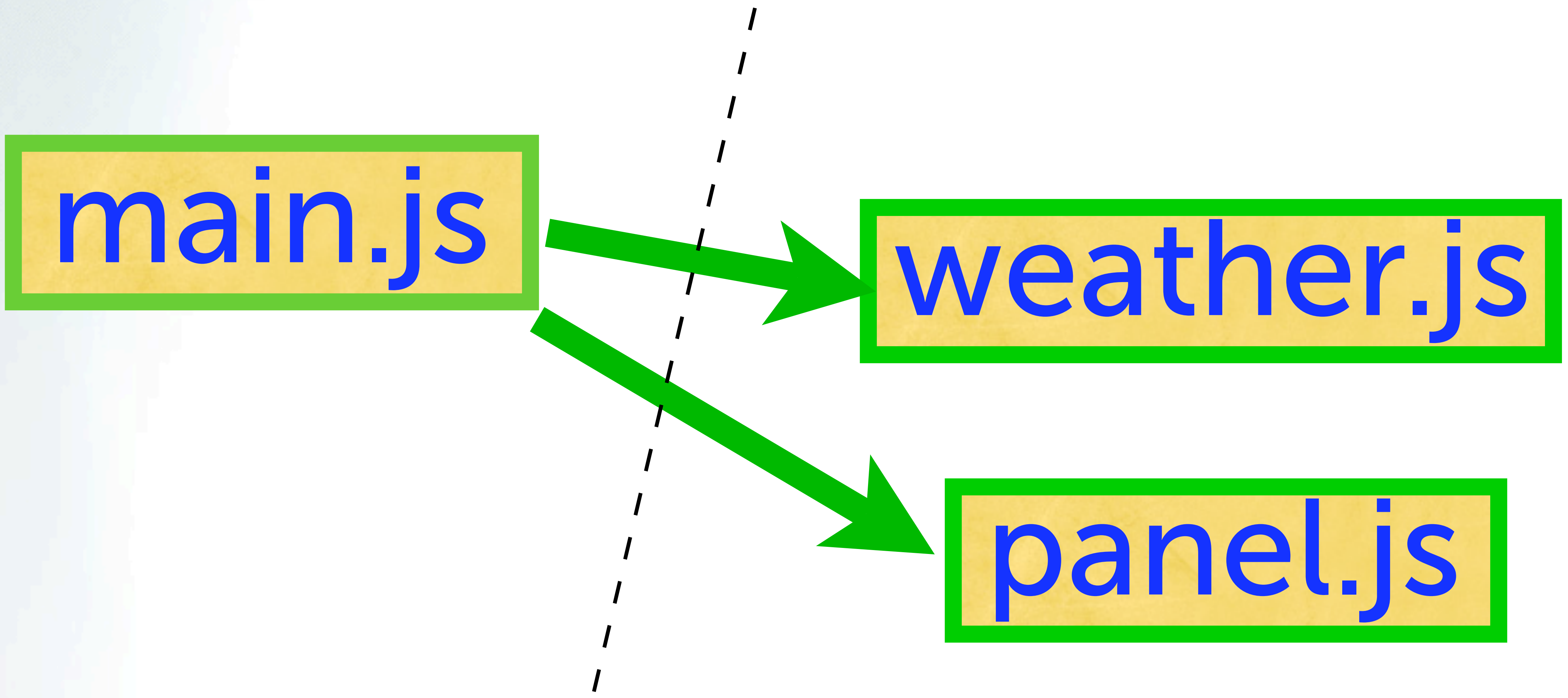
This extension can access:

- fetch weather data
- display UI panels

.. you'd have something that looks a lot like the Chrome install dialog.

The big difference is that these modules can be written by anyone, so they aren't limited by the speed at which the platform is being developed. What matters is that the module has been reviewed by someone who can do a good job explaining how much power a given module provides, and who can inspect the code for vulnerabilities that would allow it to violate that description. The user is delegating some (but not all) of their decision-making power to these reviewers. For Firefox, the AMO review team would be the default choice, but users could point that at someone else instead (perhaps their local IT department). And the choice of reviewer is just a runtime configuration control, which can be changed at will, so you can apply lots of eyeballs to the review process. Since add-ons are all about 3rd parties providing new functionality, you don't really want to be dependent upon the vendor for approval or implementation of new APIs.

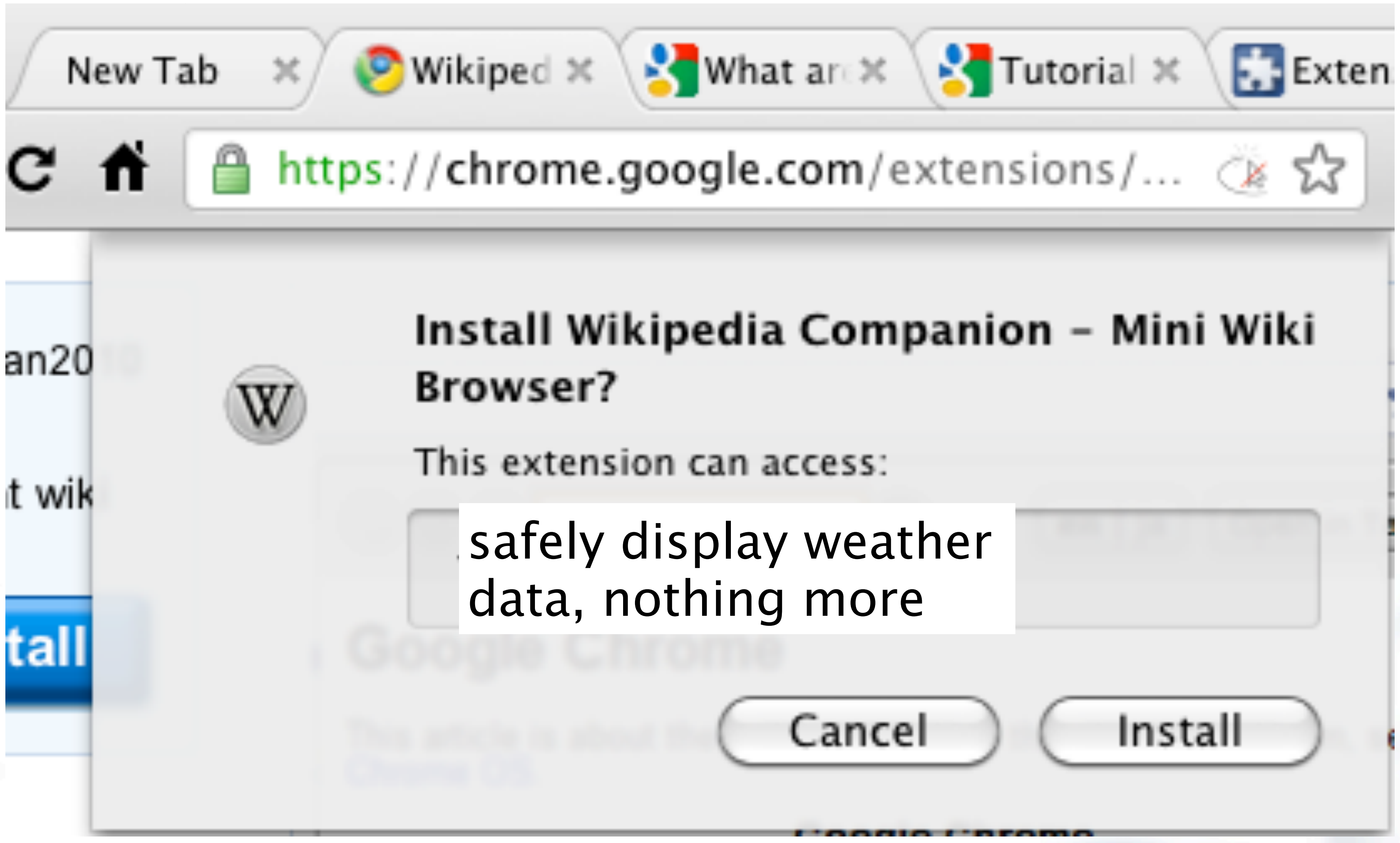
Reviewing Add-On Description



main.js

weather.js

panel.js



Install Wikipedia Companion – Mini Wiki Browser?

This extension can access:

safely display weather data, nothing more

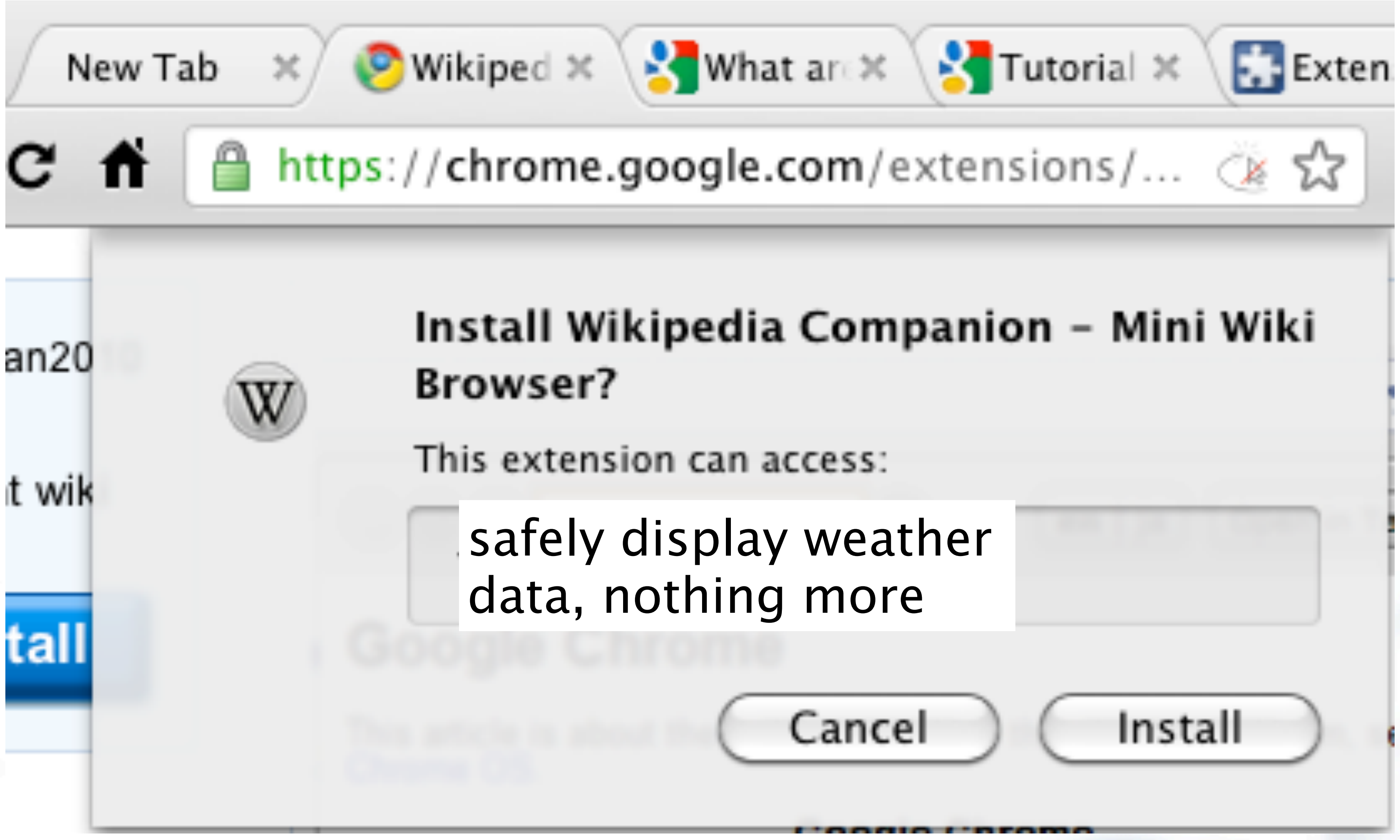
mozilla Labs Jetpack

If a reviewer also examined the top-level code, they could simply publish an assertion that the add-on’s overall description was accurate, in which case the user could rely upon just the description to make their install/no-install choice.

This is effectively what AMO is doing for Firefox add-ons now, except without the modularity or the option of reviewing anything less than the whole thing.

Reviewing Add-On Description

main.js



Install Wikipedia Companion – Mini Wiki Browser?

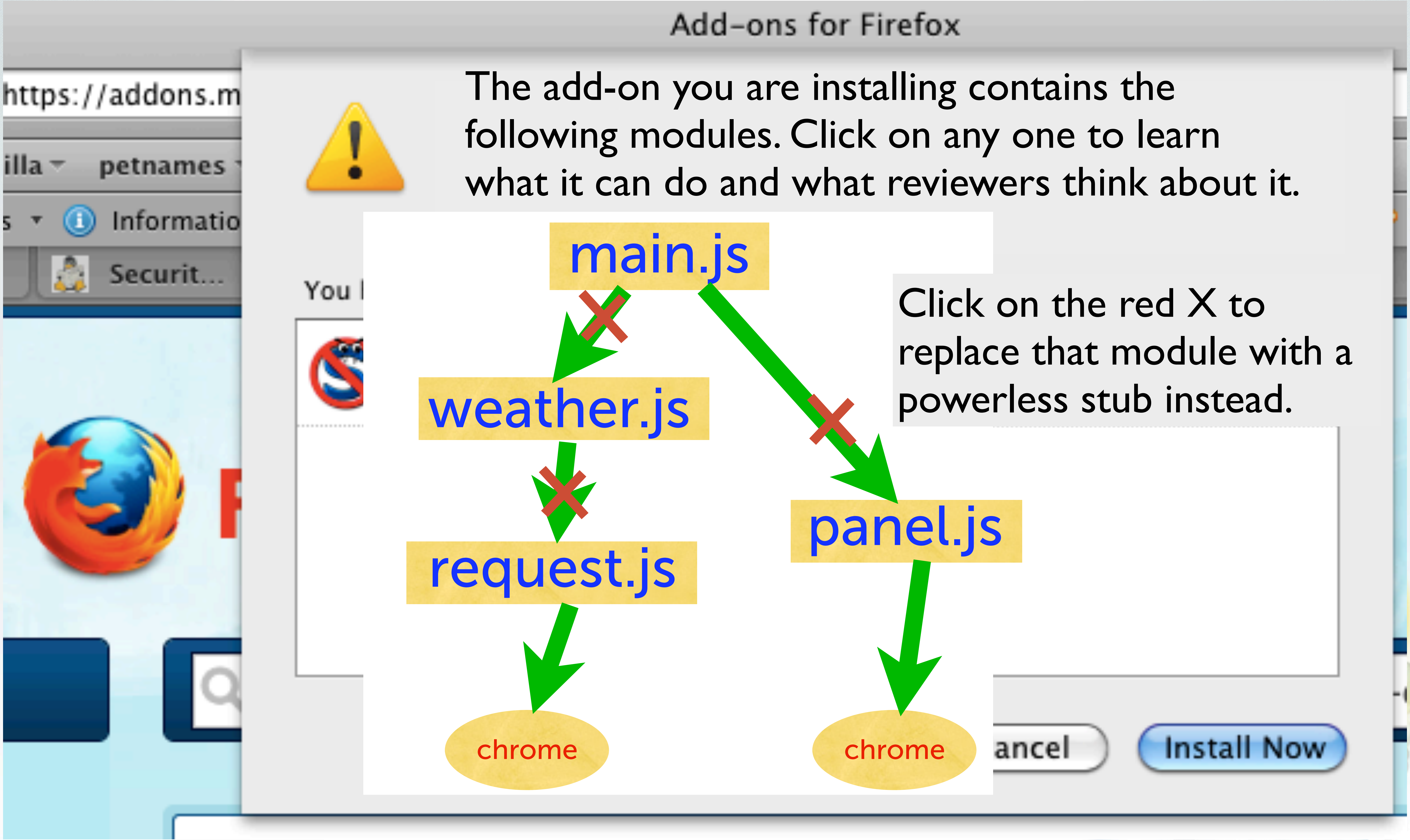
This extension can access:

safely display weather data, nothing more

If a reviewer also examined the top-level code, they could simply publish an assertion that the add-on's overall description was accurate, in which case the user could rely upon just the description to make their install/no-install choice.

This is effectively what AMO is doing for Firefox add-ons now, except without the modularity or the option of reviewing anything less than the whole thing.

Give The User More Information/Control?



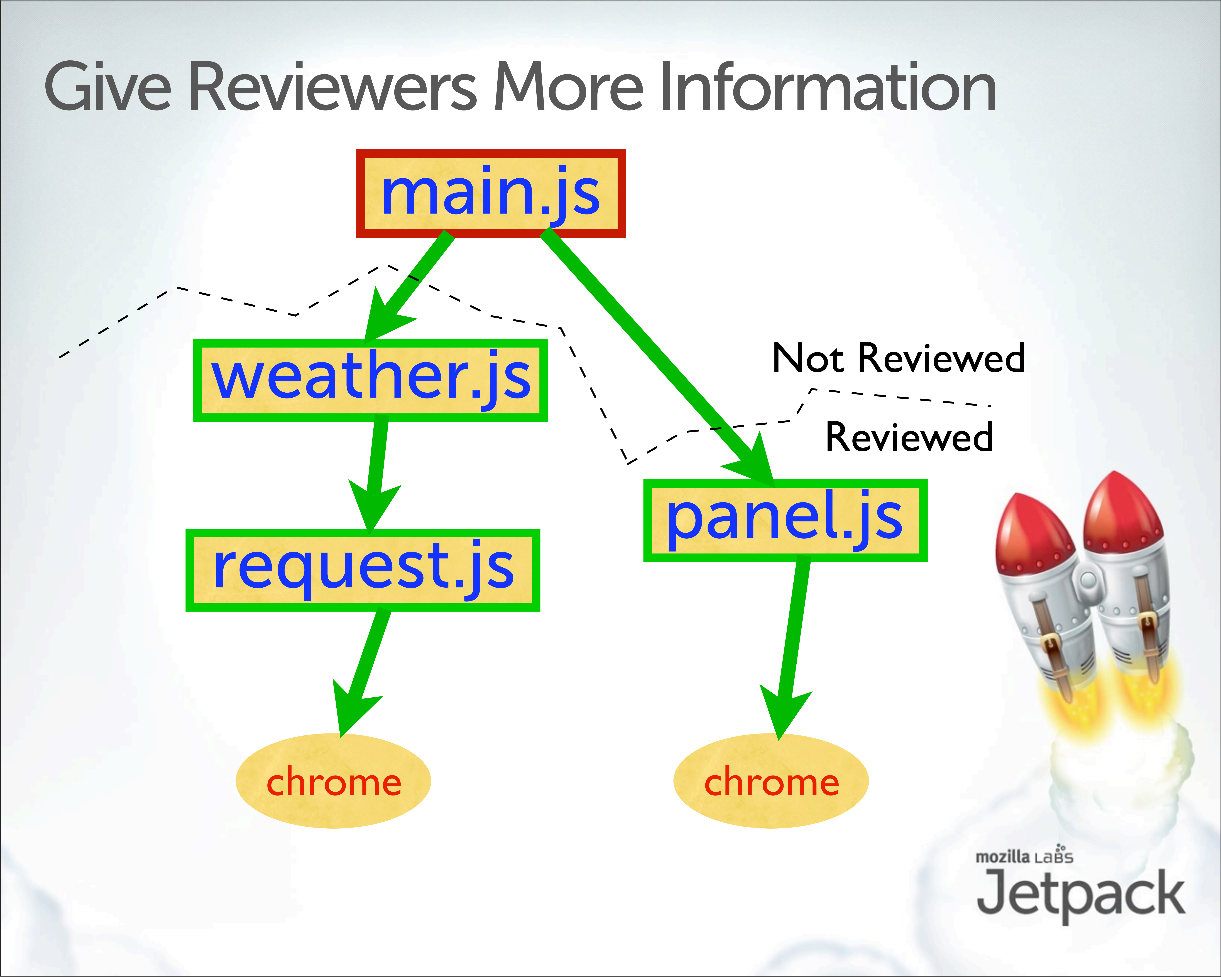
The add-on you are installing contains the following modules. Click on any one to learn what it can do and what reviewers think about it.

Click on the red X to replace that module with a powerless stub instead.

Jetpack

The add-on installer could give the user a more interactive way to learn about what the add-on can do, by inspecting individual modules. You could even imagine an installer that could alter the module graph, by replacing modules with powerless dummy stubs, or with degraded variants, like a geolocation module that returned coarse or random coordinates.

Of course, there are a lot of UX challenges with this. But I think this is a useful way to think about the problem: existing schemes are basically degenerate forms of this, which provide less information or give the user less control.



Give Reviewers More Information

main.js

weather.js

panel.js

request.js

chrome

chrome

Not Reviewed

Reviewed

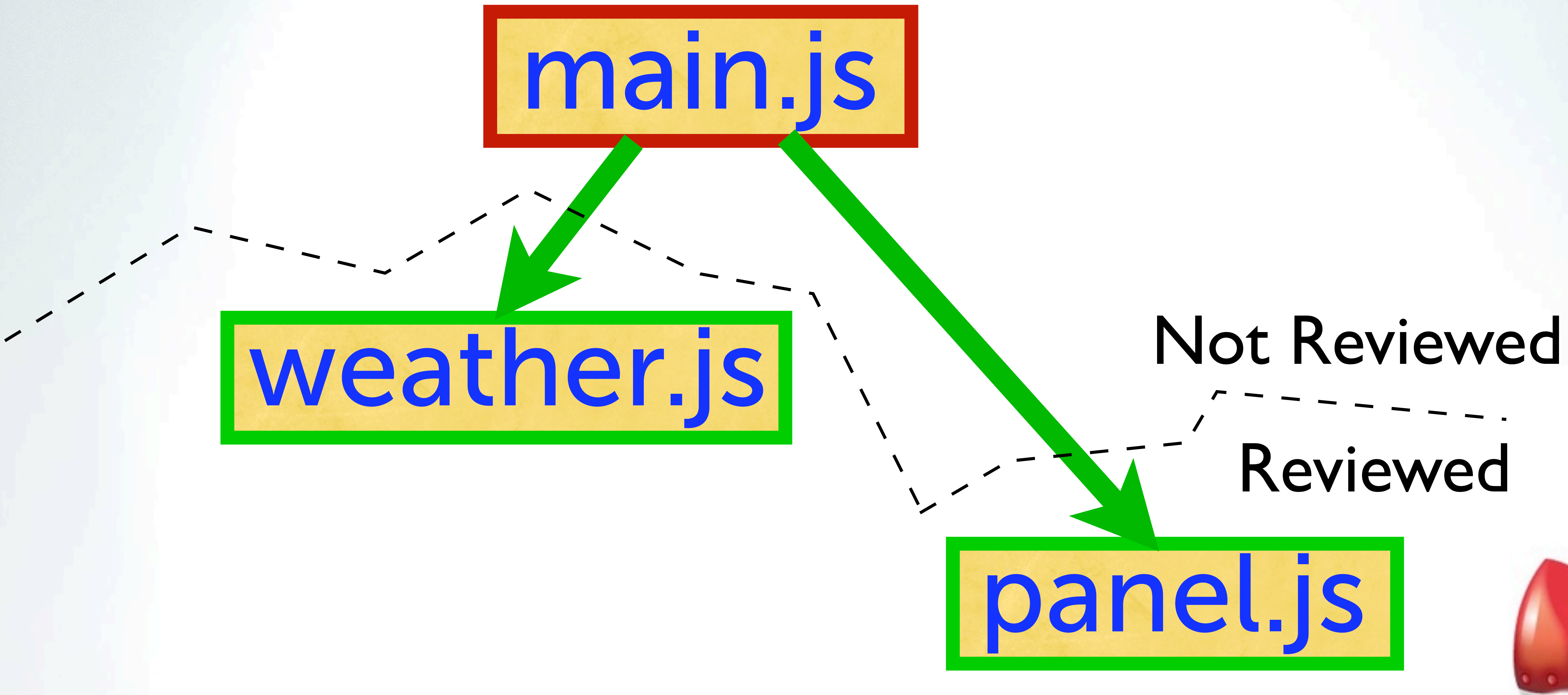
mozilla Labs Jetpack

There are a lot of intermediate points that are worth looking into here. People criticize the Chrome and Android installation dialogs for presenting too much information that most users don't know how to deal with, and then asking them the same binary Blame-The-User question as always. So maybe full user control is not the most appropriate answer.

One interesting alternative is just to provide this tree-of-module data to AMO reviewers, who *are* experts in the field, so they can spend their limited time more efficiently, but still result in a binary approve/deny decision for each add-on.

Simply providing tools to the review team which let them identify which modules they've already reviewed would be a big win.

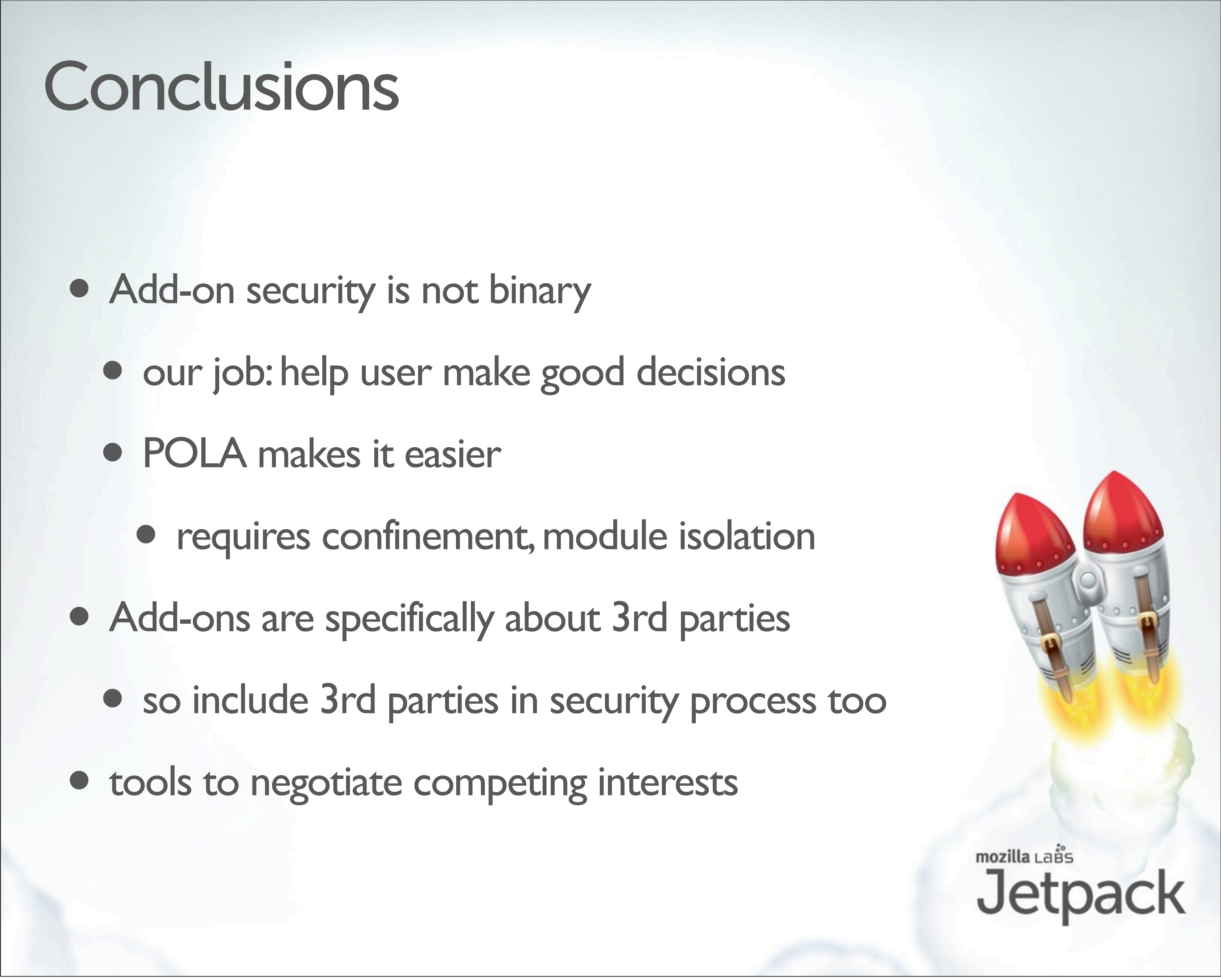
Give Reviewers More Information



There are a lot of intermediate points that are worth looking into here. People criticize the Chrome and Android installation dialogs for presenting too much information that most users don't know how to deal with, and then asking them the same binary Blame-The-User question as always. So maybe full user control is not the most appropriate answer.

One interesting alternative is just to provide this tree-of-module data to AMO reviewers, who *are* experts in the field, so they can spend their limited time more efficiently, but still result in a binary approve/deny decision for each add-on.

Simply providing tools to the review team which let them identify which modules they've already reviewed would be a big win.



Conclusions

- Add-on security is not binary
  - our job: help user make good decisions
  - POLA makes it easier
    - requires confinement, module isolation
- Add-ons are specifically about 3rd parties
  - so include 3rd parties in security process too
- tools to negotiate competing interests

mozilla LABS
Jetpack

So that kind of covers what I've been thinking in the security realm for Jetpack.

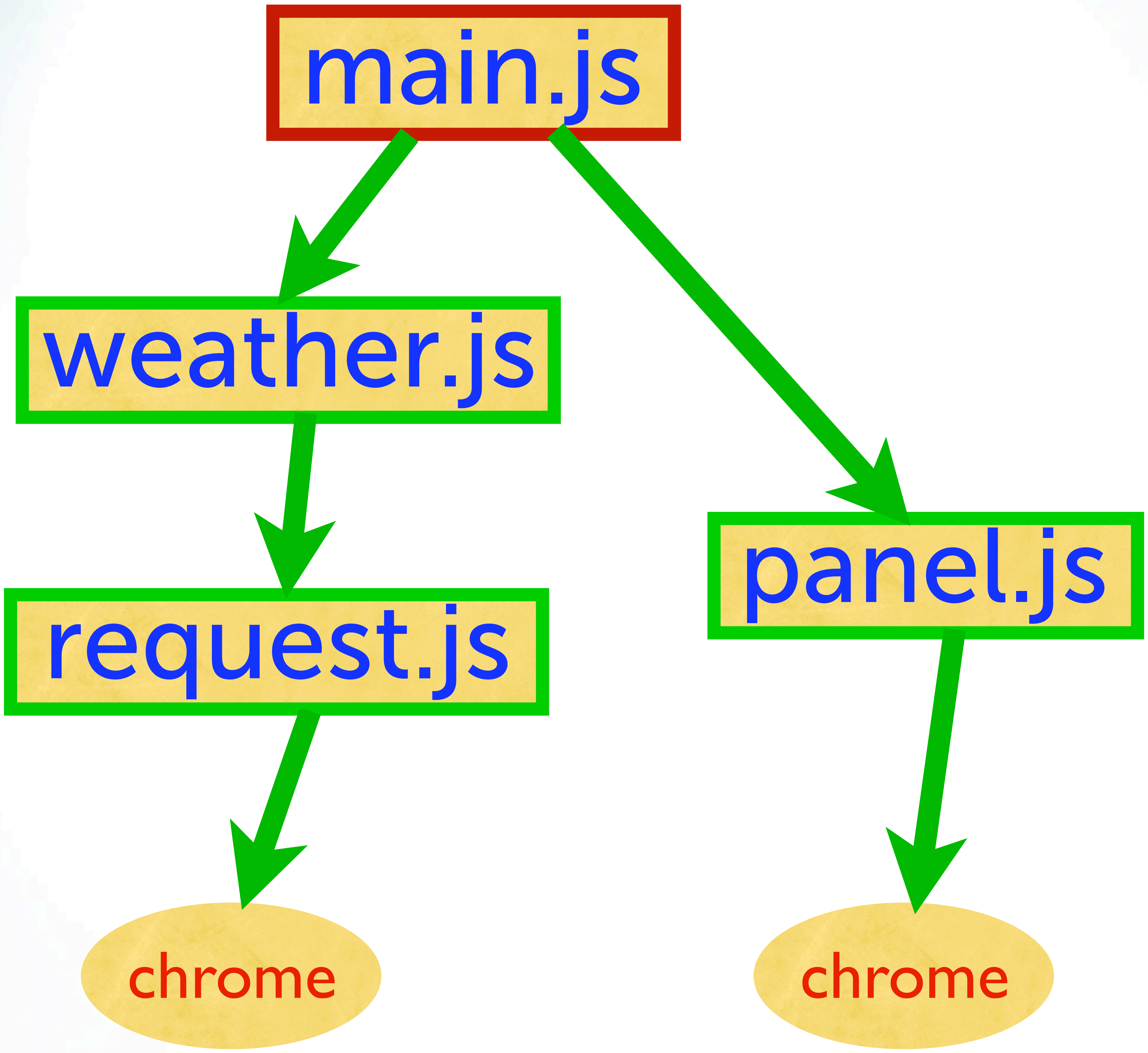
..

and finally, when you look at a module graph like .., I think the right frame of mind is to think about how to negotiate competing interests. Treat each of these modules as an anthropomorphic little person, with some private agenda that may or may not match what the callers want to do. [insert Dr. Evil's Text Editor example]. Even if the module is benevolent, it may become compromised because of a bug. If your code can survive malicious behavior on the part of these modules, it can survive accidental bugs too. When you do this right, a lot of security bugs turn into minor nuisances, and the security of the overall system is greatly improved.

Thanks!

- Jetpack
- Improving Security in Firefox Add-ons
- https://jetpack.mozillalabs.com/
- http://github.com/mozilla/addon-sdk
- Brian Warner, Mozilla Labs
- `warner@mozilla.com`

mozilla Labs
Jetpack



main.js

weather.js

request.js

panel.js

chrome

chrome

mozilla Labs
Jetpack