An Analysis of the Mozilla Jetpack Extension Framework

Rezwana Karim[1] Mohan Dhawan[1] Vinod Ganapathy[1] Chung-chieh Shan[2]

[1] Rutgers University
{rkarim,mdhawan,vinodg}@cs.rutgers.edu
[2] University of Tsukuba, Japan
ccshan@post.harvard.edu

Abstract. The Jetpack framework is Mozilla's newly-introduced extension development technology. Motivated primarily by the need to improve how scriptable extensions (also called addons in Firefox parlance) are developed, the Jetpack framework structures addons as a collection of modules. Modules are isolated from each other, and communicate with other modules via cleanly-defined interfaces. Jetpack also recommends that each module satisfy the principle of least authority (POLA). The overall goal of the Jetpack framework is to ensure that the effects of any vulnerabilities are contained within a module. Its modular structure also facilitates code reuse across addons.

In this paper, we study the extent to which the Jetpack framework achieves its goals. Specifically, we use static analysis to study *capability leaks* in Jetpack modules and addons. We implemented Beacon, a static analysis tool to identify the leaks and used it to analyze 77 core modules from the Jetpack framework and another 359 Jetpack addons. In total, Beacon analyzed over 600 Jetpack modules and detected 12 capability leaks in 4 core modules and another 24 capability leaks in 7 Jetpack addons. Beacon also detected 10 over-privileged core modules. We have shared the details with Mozilla who have acknowledged our findings.

1 Introduction

Several modern browsers support an extensible architecture that allows end-users to enhance and customize the functionality of the browser. Extensions come in a variety of flavors, such as executable plugins to interpret specific MIME formats (*e.g.,* PDF readers, ActiveX, Flash players), browser helper objects, and scriptable addons.

Our focus in this paper is on scriptable extensions for the Mozilla Firefox browser. Such scriptable extensions, also called *addons*, are written in JavaScript, are widely available, and have contributed in large part to the popularity of the Firefox browser and related tools, such as the Thunderbird mail client. As of December 2011, over 7000 addons, supporting a wide variety of functionalities, are available for Firefox via the Mozilla addons page [3]. Popular examples of addons for Firefox include GreaseMonkey [3], which customizes the look and feel of Web pages using user-defined scripts, Firebug [2], which is a JavaScript code development environment, and NoScript [10], which is a security addon that aims to prevent the execution of unauthorized third-party scripts.

[3] http://addons.mozilla.org

To support rich functionality, the browser exports an API that JavaScript code in an addon can use to access privileged browser objects and services. On Firefox, this API is called the XPCOM interface (cross-domain component object model) [26], and allows JavaScript code in addon to access a wide variety of services, such as the file system and the network. Access to the XPCOM interface endows JavaScript code in an addon with capabilities that are normally not available to JavaScript code in a Web page. For example, JavaScript code in an addon can freely send `XMLHttpRequest`s to any Web domain, without being constrained by the same-origin policy. The addon can also freely access objects stored on the file system, such as the user's browsing history, cookie store, or any other files accessible by the browser process.

Unfortunately, the privileges endowed by the XPCOM interface can be misused by attacks directed against vulnerable extensions. A recent study of over 2400 Firefox addons [14] found several addons demonstrating insecure programming practices and exploitable vulnerabilities. A successful exploit against vulnerable addons gives the attacker privileges to access the XPCOM interface, via which he can access the rest of the system.

A key problem that has contributed thus far to vulnerabilities and insecure programming of Firefox addons is *the lack of development tools for addon authors*. Addon authors have thus far been required to write their code from scratch, directly accessing the XPCOM interface to perform privileged actions. Such an approach lacks modularity, and provides too much authority to each addon. An exploitable vulnerability anywhere in the addon typically exposes the entire XPCOM interface to the attacker.

To address this problem, Mozilla has recently been developing the *Jetpack framework* [5], officially known as addon SDK [7], a new extension development technology that aims to improve the way addons are developed. It does so using *modularity* and by attempting to enforce *the principle of least authority (POLA)* [27]. A Jetpack addon consists of a number of *modules*. Each module explicitly requests the capabilities that it requires, *e.g.,* access to specific parts of the XPCOM interface, and is isolated from the other modules at the framework level, *i.e.,* its objects are not visible to other modules in the Jetpack addon unless they are explicitly exported by the module. The Jetpack framework therefore aims to contain the effects of vulnerabilities within individual modules by structuring the addon as a set of modules that communicate with each other with clearly defined interfaces, and by ensuring that each module only requests access to the XPCOM interfaces that it needs. The design of the Jetpack addon framework also facilitates code reuse: Jetpack addon authors can contribute the modules used in their addons to the community, following which others can use the modules within their own addons. To bootstrap this process, Mozilla has provided a set of *core modules* that provide a library of features that will be useful for a wide variety of addons.

In this paper, we study the extent to which the Jetpack framework achieves its goals. Specifically, we use static analysis to study *capability leaks* in Jetpack modules and addons. A capability leak happens when a module requests access to a specific XPCOM interface (*i.e.,* a capability), and inadvertently exports a pointer to this interface. Capability leaks allow other modules to access this XPCOM interface (via the exported pointer) without explicitly requesting access to the interface, thereby violating modularity. We also use the same static analysis to study *violations of POLA*, *i.e.,* cases where a module requests access to an XPCOM interface, but never uses it. A vulnerable mod-

ule that violates POLA can endow an attacker with more privileges than if the module satisfied POLA.

We applied our analysis to a corpus of over 600 Jetpack modules, which include 77 core modules from Mozilla's Jetpack addon framework and 359 Jetpack addons. Our results show that there are 12 capability leak in 4 core modules and another 24 capability leaks in 7 Jetpack addons. Our analysis also detected 10 core modules violating POLA by requesting privileged resources that they do not utilize. We have shared the details with Mozilla who have acknowledged our findings for the core modules.

2 Background and Motivation

The Jetpack framework [5] focuses on easing the extension development process with an emphasis on modular development, code sharing and security. The framework provides high-level APIs, allowing addon authors the ease of writing extensions using standard Web technologies, like JavaScript and CSS. This is in contrast with traditional extension development, which required developers to be proficient in Mozilla specific technologies like XUL [12] and XPCOM [26].

A Jetpack addon is a hierarchical collection of JavaScript modules, with each module exporting some key functionality. A typical Jetpack addon consists of core modules, user modules and some glue code. Core modules provide low-level functionality and are provided by Mozilla itself. User modules are usually authored by the addon developer or other third-parties who have contributed their code to the community. Glue code ties up all the modules to provide the expected functionality of the addon. On execution, the Jetpack runtime *loads* each component module in a separate sandboxed environment resulting in namespace separation for code within the modules. Inter-module communication is facilitated by special JavaScript constructs, `exports` and `require`, which serve as well-defined entry and exit points for the modules. The `exports` interface enables a module to expose functionality by attaching properties to the `exports` object. The `require` function enables a module to import such exported functionality.

The guidelines by Mozilla advise developers to follow the principle of least authority (POLA) [8] when designing modules. This helps in attenuating the capabilities of modules. The modular architecture of a Jetpack addon coupled with strong isolation between the modules helps to confine the effects of module execution. This is in sharp contrast to the traditional extension development model, where monolithic extensions shared the same namespace and had privileged access to large number of resources via the XPCOM interface. Prior work [13, 15, 20] has shown that such extensions are vulnerable to a variety of security threats.

Although not recommended, a Jetpack module may also directly invoke XPCOM interfaces if the desired functionality is not exported by either the core or user modules. However, this is dangerous since interaction with XPCOM interfaces provides access to privileged resources and inexperienced addon authors could inadvertently attach such capabilities to the `exports` interface. Importing such modules would make the requesting module over-privileged and violate POLA.

Figure 1 shows the architecture of a simple Jetpack addon which enables the user to download files from the Web. Each of the dotted boxes in the figure represents a module. Modules such as file, network, preferences represent the core modules and are provided

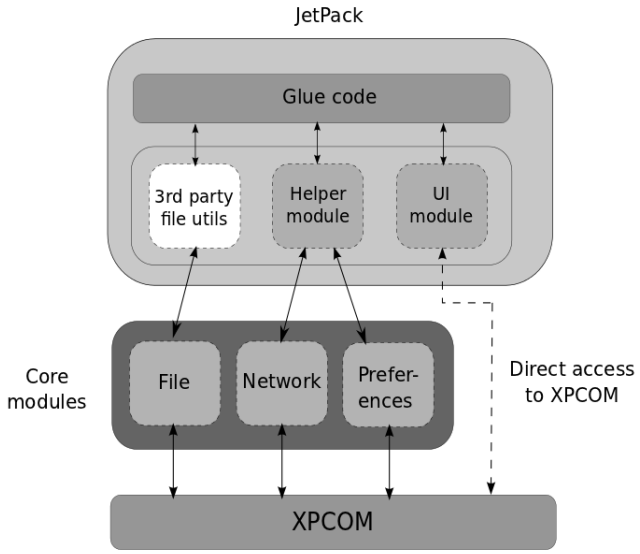

Fig. 1: Structure of a simple Jetpack addon.

by Mozilla. The user-level modules include the helper module, the UI module and third-party file utilities. As shown in the figure, the helper module and file utilities build on top of the services exported by the core modules. The UI module directly invokes XPCOM interfaces to support functionality not provided by core modules, such as user alerts or dialog boxes. Although such direct invocations are not recommended (as shown by the dotted line), they are allowed till the Jetpack framework matures and Mozilla develops core modules for all key services.

Capability leak in a Jetpack addon

Consider the code snippet as shown in Figure 2 which represents the actual code of the `Preferences` module from 'Customizable Shortcuts' [1], a popular Jetpack addon with over 5000 users. This module exports a method `getBranch` which inadvertently enables access to the browser's entire preference tree. If another module imports the `Preferenecs` module, it would receive additional capabilities to access and modify the user's preferences for all extensions *without explicitly requiring access to the user preferences*; in effect the importing module becomes over-privileged. Although the Jetpack framework recommends adherence to POLA, it does not safeguard against developer mistakes, with the result that unintended capability leaks are frequent.

Let us now examine the code in detail to understand the cause of the capability leak. In line 1, the module requests `chrome` authority to enable it to access ***any***

```
(1)  const {Cc, Ci} = require("chrome");
(2)  let Preferences = {
(3)    _branches: {},
(4)    _caches: {},
(5)    getBranch: function (name) {
(6)      if (name in this._branches) return this._branches[name];
(7)      let branch = Cc["@mozilla.org/preferences-service;1"]
           .getService(Ci.nsIPrefService).getBranch(name);
(8)      .../* other statements */
(9)      return this._branches[name] = branch;
(10)   }, ... /* other properties */
(11) };
(12) exports.Preferences = Preferences;
```

Fig. 2: Code snippet of a module from a real-world Jetpack addon which leaks the capability to access and modify browser preferences.

XPCOM interface. Line 2-11 declare a `Preferences` object with several properties (including `_branches` and `getBranch`) defined on it. On line 12, the module exports the `Preferences` object by attaching it to the `exports` object. Since the entire `Preferences` object is exported, a module which `requires` this module would have access to all its properties, including `getBranch`.

The `getBranch` method utilizes the `chrome` privileges acquired in line 1 to first create an instance of the XPCOM interface `nsIPrefService` and then invoke the `getBranch` method defined on the interface. The `getBranch` method returns an instance of another XPCOM interface `nsIPrefBranch`, which provides a handle to access and modify user preferences. After the assignment in line 7 is complete, `branch` stores an instance of `nsIPrefBranch`. In line 9, the method returns this privileged instance to the caller. Thus, the capability to manipulate the preference tree is leaked through the `exports` interface of the module.

The capability leak from `Preferences` module thus makes an importing module over-privileged, thereby violating POLA. Such a capability leak might even cause inadvertent deletion of user preferences. Ideally, the module should have been designed in a manner to either export access only to its own preference branch, or return primitive values corresponding to the preferences rather than a reference to the branch.

The module also violates another Jetpack addon design principle, which is to utilize capabilities of core modules whenever possible and maintain the hierarchical module structure. The `Preferences` module accesses and returns a reference to the preferences XPCOM interface even though the core modules provide equivalent functionality through the `preferences-service` module, thereby breaking the expected hierarchical structure. The absence of any restriction on developers to use core modules only exacerbates the problem.

Failure to adhere to Jetpack addon guidelines and principles is common in Jetpack modules, in part due to the absence of functionality in core modules and also because of the available choices during module design and implementation. Although adherence to POLA ensures that a module has the minimal set of capabilities required to perform

Entity	Sensitive attributes and methods
Bookmarks	nsIRDFDataSource
Chrome	Components.classes, Components.interfaces, Components.utils, Components.result
Cookies	nsICookieService, nsICookieManager
Document	window.gBrowser.contentDocument, window.document
Files	nsILocalFile, nsIFile
Passwords	nsIPasswordManager, nsIPasswordManagerInternal.
Preferences	nsIPrefService, nsIPrefBranch
Serivces	nsIIOService, nsIObserverService, nsIPromptService
Streams	nsIInputStream, nsIFileInputStream
Window	nsIWindowMediator, nsIWindowWatcher
XPCOMUtils	nsIModule, generateQI

Table 1: List of some privileged resources and their access interfaces.

its desired functionality, it is hard to implement in practice due to developer mistakes and refactoring oversights. A capability leak analysis for Jetpack modules would help to identify modules that violate POLA and restrict any security threat only to the concerned module.

3 Static analysis of Jetpack modules and addons

In this section we describe a static analysis to detect sources of capability generation in Jetpack modules, flow of capabilities through a module and across the module interface.

The capability leak analysis is an instance of static information flow tracking where taint is modeled as the capability of accessing sensitive sources. A list of the sensitive sources considered in our analysis is given in Table 1. These sources are classified as sensitive as they allow module code to access browser resources and perform privileged operations, such as access to arbitrary DOM elements, read/write access to the cookie and password stores, unrestricted access to the local file system and the network, etc.

In the context of Jetpack modules, an object acquires capabilities if (a) it directly accesses any of the sensitive sources (XPCOM interfaces) or (b) aliases capabilities inherited by the module via an explicit `require` call. In our analysis, an object is marked *privileged* if it directly acquires capabilities, while it is considered *tainted* if it transitively acquires the capabilities.

Both privileged and tainted objects propagate the associated capability through different program paths and can potentially leak it through the module's `exports` interface. Thus, the `exports` interface of each module is an information sink. A module can leak capabilities if it exports:

- direct references to privileged or tainted objects, and/or
- functions that provide references to privileged or tainted objects on invocation or on construction.

To identify capability leaks through module interfaces, we do a flow- and context-insensitive call-graph based static analysis of JavaScript in the module code. Our analysis converts the JavaScript code into the Static Single Assignment (SSA) [19] form and analyzes each SSA instruction. It then processes these facts to perform capability leak analysis. The analysis obtains a degree of flow sensitivity by performing a flow insensitive analysis on an SSA representation of the program.

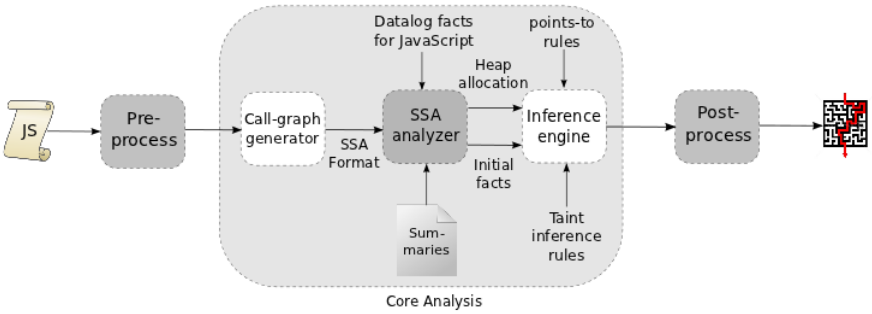

Fig. 3: Overall workflow of our analysis.

Our analysis models taint values to flow *upwards* in an object hierarchy i.e. an object is tainted if it itself is tainted or any of its properties are tainted. The key insight is that properties can be accessed given a reference to the parent object but not vice-versa. Thus, for the code snippet in Figure 2, `_branches` in line 9 is tainted because one of its properties is assigned to `branch`, which is privileged (line 7). Similarly, `Preferences` also gets tainted as one of its children (`_branches`) is tainted. Since there was no capability assignment to `_caches`, it remains untainted.

We have adopted a conservative approach to handle arrays. Since it is statically impossible to precisely determine the index for every array load, store, or access instructions, if any element in the array is tainted then the entire array is marked tainted. Unlike objects, our analysis models all array properties to be tainted if any of the siblings is tainted.

The analysis is inter-procedural. It models functions call sites, arguments and captures the appropriate flow of taint across function invocations. Primitive values are not modeled. Our analysis also does not implement any string analysis. This could affect capability flows arising from string manipulation and dynamic constructs like `eval`. JavaScript containing `eval` is supported, however the code introduced by `eval` is not modeled.

3.1 Stages of the analysis

Our analysis is based upon Datalog and proceeds in three stages. In the first stage, the analysis pre-processes the addon code to make it amenable to static analysis. The next stage performs the core analysis on the pre-processed code. The core analysis generates Datalog facts that represent capability flow in the Jetpack addon code. The results of core analysis are then processed in the third stage to identify offending flows in the source code of the Jetpack addon. Figure 3 illustrates a schematic diagram of the analysis of a Jetpack addon. The components gray are contributions of this work, while those in white are off-the-shelf tools.

We now describe the various stages of the analysis in detail:

Construct	Desugared to	Code	Desugared Code
Destructuring assignment	property access and assignment	`var {Cc, Ci} = require("chrome");`	`var Cc = require("chrome").Cc;` `var Ci = require("chrome").Ci;`
`let` †	`var`	`let foo = 5;`	`var foo = 5;`
`const`	`var`	`const SIZE = 100`	`var SIZE = 100`
lambda Function	Function	`f(x) x * x`	`f(x) { return x * x; }`

Table 2: Pre-processed JavaScript constructs and their desugared forms.† We desugar all forms of `let` i.e. statement, expression and definition.

3.1.1 Pre-processing: Our core analysis (as will be described in Section 3.1.2) is based on call-graph construction. The pre-processing stage process the module code to facilitate construction of a complete call-graph for the module.

Since functions are first-class objects in JavaScript and can be properties of other objects, it is possible that such functions are never invoked within the module. Further, if these functions are exported by the module, they could be invoked by the module requesting them. A call-graph generated for such a module would be incomplete since it would not reflect invocations for all the functions. Therefore, we append the module code with additional JavaScript code which would enable the call-graph generator to invoke all functions and generate the complete call-graph for the module. To do so, we consider all functions and properties (including JavaScript getters and setter) reachable from the module's `exports` interface and append appropriate JavaScript statements for their invocation.

We do not append function objects defined in event handling or callback code because the Jetpack runtime freezes the `exports` interface when the module has finished loading. This restricts all event handlers from attaching or modifying `exports` interface. However, the loader does not perform deep freeze of the `exports` object making it possible to modify any property reachable from the interface. Beacon may therefore have false negatives. We plan to extend Beacon to analyze all event handlers.

The pre-processing stage is also required to make the Jetpack addon code amenable for static analysis. To do so, we desugar some of the JavaScript constructs into simpler forms. For example, 'destructuring assignment' is a popular JavaScript construct that mirrors the construction of array and object literals. In essence it only represents syntactic sugar to extract data from arrays or objects. As part of pre-processing, we desugar it and convert it to statements involving simple property access and assignment. For other constructs like `let` and `const`, we change them to `var` statements while keeping the semantics unchanged. Table 2 lists set of the pre-processed constructs along with their desugared forms.

The pre-processing stage also includes code re-writing to simplify statements involving Mozilla specific XPCOM [26] interfaces, which indicate creation or access of privileged resources. To do so, we replace all such XPCOM instances by stubs indicating function calls. For example, the statement in line 7 of Figure 2 is re-written as shown below:

```
let branch = Cc["@mozilla.org/preferences-service;1"]      →     let branch = MozPrefService()
           .getService(Ci.nsIPrefService).getBranch(name);                    .getBranch(name);
```

Entity	Type	Capability
`exports`	Object	`prefBranch`
`exports.Preferences`	Object	`prefBranch`
`exports.Preferences._branches`	Object	`prefBranch`
`exports.Preferences.getBranch`	Function	`prefBranch`

Table 3: Summary of `preferences` module showing the capability leak.

We also create summaries to indicate capabilities accessible from the stub methods. This summary is fed to the analysis engine to enable it to accurately model the flow of capabilities when handling code that accesses properties on the stub method. For example, the module summary of `MozPrefService` would have one entry for `getBranch` which returns the capability `PrefBranch`.

3.1.2 Core analysis: For the purpose of statically analyzing the pre-processed JavaScript code we use an off-the-shelf tool to generate a call-graph in the SSA format. We then generate appropriate Datalog facts corresponding to statements in the JavaScript code and apply inference rules for points-to and capability flow analysis.

Our points-to analysis is inspired by the JavaScript points-to analysis introduced in Gatekeeper [22]. The key distinction is that in our analysis, all program variables carry taint information as well, thereby performing capability flow analysis together with points-to analysis. Similar to prior works [22, 30], we adopt a relatively standard way to represent a program as a database of facts. The set of Datalog relations deployed for the analysis are summarized in Table 4. Each of these relations is of fixed type and arity. The relations specify how points-to and taint information are propagated. We represent heap-allocated objects and functions using the alphabet H, program variables by V, fields by F, call sites by I, integers by Z and capabilities by P.

Unlike prior works [13, 22] which perform whole program analysis, our analysis focuses on modular JavaScript code, such as Jetpack modules. Analysis of individual modules requires that capabilities of each module be appropriately seeded based on which other modules it imports. Since invoking functions from an imported module is akin to using library or foreign functions, we model such functionality as a summary of each module. Thus, a comprehensive analysis of a particular module requires that the summary of each of the imported modules be fed to the analysis engine.

Our analysis focuses primarily on detecting capability flows, thus our summaries only reflect capability leaks possible through the module's `exports` interface, A module's summary typically contains information about the properties of the `exports` interface, their types and taint values reflecting the capabilities associated with the object. Table 3 shows the summary for the code module shown in Figure 2.

Our module summaries simply list the capabilities exported by specific properties exported by a module. In JavaScript, functions can also be exported. However, our summaries are currently not parameterized by the arguments to such functions, which may lead to false negatives in our analysis.

Once summaries for all the imported modules are available, the analysis engine constructs a call graph along with the control-flow graphs for each method in the module to be analyzed. These control-flow graphs consist of several basic blocks which comprise

Relations for points-to analysis		
Heap mapping	ptsTo(V, H)	represents a points-to relation for a variable
	heapPtsTo(H_1, F, H_2)	represents points-to relation for heap objects
	prototypeOf(H_1, H_2)	record object prototype
Object manipulation	assign(V_1, V_2)	represents variable assignments
	store(V_1, F, V_2)	represents field store for an object
	load(V_1, V_2, F)	represents field load from an object
Function manipulation	calls(I, H)	represents call site I invoking method M
	formal(H, Z, V)	represents formal argument of method M
	methodRet(H, V)	represents return value of a method
	actual(I, Z, V)	represents actual parameter of a call site
	callRet(I, V)	represents return value for a call site
Relations for capability flow analysis		
Capability flow	isPrivileged(H, P)	indicates heap object H is privileged with type P
	isTainted(H, P)	indicates heap object H is tainted with type P
	idIsPrivileged(V, P)	indicates variable V is privileged with type P
	idIsTainted(V, P)	indicates variable V is tainted with type P

Table 4: Datalog relations used in our static analysis.

of SSA statements. The analysis engine traverses each of these statements and produces Datalog facts capturing its semantics, as illustrated in Table 5. It also generates heap allocation mappings for the objects and functions, denoted by h_{fresh}. During this phase, several Datalog facts corresponding to the relations shown in Table 4 are generated. The analysis engine then applies the Datalog inference rules presented in Table 6 over the initial set of facts to keep track of aliases and the flow of capability through the JavaScript code.

3.1.3 Post-processing: The combination of initial set of Datalog facts and facts generated after the application of inference rules abstract the behavior of the Jetpack module under analysis. These facts provide information regarding capability flows for the module being analyzed. The post-processing stage links this information back to the source code, identifying possible locations in the source code where capabilities were generated and the properties of the `exports` interface through which they were externalized. This processed information is also utilized for generating a summary for the analyzed module.

3.2 Capability flow: A concrete example

We now demonstrate how the analysis detects capability flows from the `exports` interface of Jetpack addon modules. Figure 4 represents a pre-processed module and the initial set of points-to facts generated by the analysis.

The pre-processed module indicates the use of capabilities within the module by the stub function `MozPrefService`. The ptsTo relations represent object allocations in the heap for each object or function declaration. The analysis engine generates a call-graph

Statement	Example Code	Generated Facts
ASSIGNMENT	$v_1 = v_2$	assign(v_1, v_2)
RETURN	return v	callRet(v)
OBJECT LITERAL	v = {}	ptsTo(v, h_{fresh})
STORE	$v_1.f = v_2$	store(v_1, f, v_2)
LOAD	$v_1 = v_2.f$	load(v_1, v_2, f)
FUNCTION DECLARATION	v = function(v_1, v_2, ..., v_n)	ptsTo(v, h_{fresh}) heapPtsTo(h_{fresh}, prototype, p_{fresh}) for z ∈ 1...n, generate formal(h_{fresh}, z, v_z) methodRet(h_{fresh}, v)
OBJECT CONSTRUCTION	v = new v_0(v_1, v_2, ..., v_n)	ptsTo(v, h_{fresh}) prototypeOf(h_{fresh}, d) :- ptsTo(v_0, h_{method}), heapPtsTo(h_{method}, prototype, d) for z ∈ 1...n, generate actual(i, z, v_z) callRet(i, v)
FUNCTION CALL	v = v_0(v_{this}, v_1, v_2, ..., v_n)	ptsTo(v, h_{fresh}) for z ∈ 1...n, this, generate actual(i, z, v_z) callRet(i, v)

Table 5: Datalog facts generated for each JavaScript statement.

Basic rules		
ptsTo(V_1, H)	:-	ptsTo(V_2, H), assign(V_1, V_2)
ptsTo(V_2, H_2)	:-	load(V_2, V_1, F), ptsTo(V_1, H_1), heapPtsTo(H_1, F, H_2)
heapPtsTo(H_1, F, H_2)	:-	store(V_1, F, V_2), ptsTo(V_1, H_1), ptsTo(V_2, H_2)
Call graph		
calls(I, H)	:-	actual(I, 0, V), ptsTo(V, H)
Inter-procedural assignments		
assign(V_1, V_2)	:-	calls(I, H), formal(H, Z, V_1), actual(I, Z, V_2)
assign(V_2, V_1)	:-	calls(I, H), methodRet(H, V_1), callRet(I, V_2)
Prototype handling		
heapPtsTo(H_1, F, H_2)	:-	prototypeOf(H_1, H), heapPtsTo(H, F, H_2).
prototypeOf(O, H)	:-	heapPtsTo(M, prototype, P), heapPtsTo(M, prototype, H), prototypeOf(O, P)
Taint propagation		
isTainted(H_1, P)	:-	heapPtsTo(H_1, F, H_2), isPrivileged(H_2, P)
isTainted(H_1, P)	:-	heapPtsTo(H_1, F, H_2), isTainted(H_2, P)
idIsTainted(V, P)	:-	ptsTo(V, H), isPrivileged(H, P), not(idIsPrivileged(V, P))
idIsTainted(V, P)	:-	ptsTo(V, H), isTainted(H, P)

Table 6: Datalog inference rules for points-to analysis.

	Pre-processed JavaScript statements	Generated Datalog facts
(1)	`var exports = {};`	ptsTo($v_{exports}$, $h_{exports}$)
(2)	`var Preferences = {`	ptsTo($v_{Preferences}$, $h_{Preferences}$)
(3)	`  _branches: {},`	ptsTo($v_{\_branches}$, $h_{\_branches}$) store($v_{Preferences}$, _branches, $v_{\_branches}$).
(4)	`  getBranch: function (name) {`	ptsTo($v_{\_branches}$, $h_{\_branches}$). store($v_{Preferences}$, getBranch, $v_{getBranch}$)
(5)	`    var branch = MozPrefService().getBranch(name);`	ptsTo(v_{branch}, $h_{prefBranch}$).
(6)	`    return this._branches[name] = branch;`	store($v_{\_branches}$, –, v_{branch})
(7)	`  }, ... /* other properties */`	
(8)	`};`	
(9)	`exports.Preferences = Preferences;`	ptsTo($v_{exports}$, $h_{exports}$) store($v_{exports}$, preferences, $v_{Preferences}$)

Fig. 4: Example showing the flow of capabilities through the module's `exports` interface.

with invocation for all methods reachable from the `exports` interface to determine the capabilities flowing out of the module. In the example, the analysis invokes the `exports.Preference.getBranch` method. For brevity, we omit the details of the invocation itself and the associated facts generated for the relevant statements.

The analysis detects capability leaks from the module by determining whether `exports` is tainted or not. To do so, it must answer the following Datalog query:

$$\mathsf{idIsTainted}(v_{exports}, \mathsf{X})?$$

where $v_{exports}$ is the SSA representation for the `exports` interface and X is the capability being exported.

Instead of operating on SSA representations, the analysis transforms the above Datalog query to operate on heap allocation representation. Thus, the new query to be resolved is:

$$\mathsf{isTainted}(h_{exports}, \mathsf{X})?$$

where $h_{exports}$ represents heap allocation for $v_{exports}$.

When the analysis invokes the `getBranch` method and analyzes line 5, it reads the summary for `MozPrefService`. This summary lists `getBranch` as method that returns the capability `PrefBranch`. Thus, the analysis engine allocates a heap object ($h_{prefBranch}$) for `nsIPrefBranch` and generates the fact: isPrivileged($h_{prefBranch}$, prefBranch). At line 6, v_{branch} holds the return value of the function `MozPrefService.getBranch(name)`, and thus v_{branch} points to $h_{prefBranch}$. For sake of brevity, we omit the processing of the `return` statement.

On consulting the Datalog inference rules in Table 6 and existing facts, the analysis infers that $h_{prefBranch}$ is stored in the heap allocation object $h_{\_branches}$ thus tainting $h_{\_branches}$. As mentioned earlier in the section, taints propagate upwards in an object hierarchy. Thus the capability `PrefBranch` flows from $h_{\_branches}$ to the heap allocation of the parent object, $h_{Preferences}$ and generates the fact: isTainted($h_{Preferences}$, prefBranch). This in turn generates a similar fact: isTainted($h_{exports}$, prefBranch). Coupled with the fact that $v_{exports}$ points to the heap allocation $h_{exports}$, the analysis resolves X to be

`PrefBranch` and determines `PrefBranch` as the capability flowing out of the module through the `exports.Preferences.getBranch` method.

4 Implementation

We realized the analysis described in Section 3 in a tool named Beacon. Beacon is built atop WALA [29], an existing static analysis tool, and uses WALA's capabilities to convert pre-processed JavaScript code into an SSA-based register-transfer intermediate representation (IR) and generate appropriate control-flow graph. Beacon analyzes each IR to generate corresponding Datalog facts, which are processed using the DES Datalog query engine [16]. The core analysis in Beacon was implemented in about 2.8K lines of Java code while an additional 700 lines of scripts were required for pre- and post-processing.

5 Results

We evaluated the effectiveness and accuracy of Beacon in detecting capability leaks by analyzing the entire set of 359 Jetpack addons and 77 core modules available to us at the time of writing the paper. In total, Beacon analyzed over 600 modules consisting of over 68K lines of JavaScript code. The performance of Beacon's static analysis heavily depends on the size of the analyzed module. On average, Beacon takes a couple of minutes and consumes 200MB per module. For the largest module (`tab-browser.js`/25KB), Beacon took 30mins and 243MB of memory. In Section 5.1 we present results from analysis of the capability leaks in core modules and Jetpack addons. In Section 5.2 we study the nature and usage of capabilities in various Jetpack addons. Lastly, in Section 5.3 we report on the use of Beacon to analyze the privileges associated with Jetpack addons and the core modules to detect over-privileged modules.

Our evaluation methodology involved pre-processing the modules to desugar any incompatible JavaScript constructs and append additional JavaScript code to ensure complete code coverage (see Section 3.1 for details). Each pre-processed module file was individually analyzed by Beacon to generate appropriate Datalog facts that were later processed to extract information about capability leaks. The post-processing also generated a summary for the module that was utilized for analysis of another modules which imported it.

5.1 Capability leaks

Beacon detected 12 capability leaks in four core modules and another 24 leaks in seven Jetpack addons. Most of the detected leaks were subtle and hard to catch through manual code review. This is reinforced by the fact that Beacon managed to detect 12 capability leaks in production quality code which has undergone numerous code reviews and has a relatively stable code base. For each of the reported leaks, we manually verified the results and observed no false positives. We shared the details of our findings with Mozilla who acknowledged capability leaks in the four core modules. Tables 7 and 8 summarize the findings.

Core module	Capability	Leak mechanism	Essential
tabs/utils †	Active tab, browser window and tab container	Function return	Yes
window-utils †	Browser window	Function return	Yes
xhr	Reference to the `XMLHttpRequest` object	Property of `this` object	No
xpcom	Entire XPCOM utility module	Exported property	No

Table 7: List of capability leaks observed in the core modules. † indicates multiple reference leaks.

Jetpack addon	Capability	Leak mechanism	Essential
Bookmarks Deiconizer	Entire XPCOM services module	Exported property	No
Browser Sign In	window, document	Return from exported function	No
Customizable Shortcut	nsIPrefBranch, nsIAtomService	Property of `this` object	No
	window	Return of function attached to `this`	No
Firefox Share	nsIPrefBranch, window	Property of `this` object	No
	Reference to built-in SQLite database	Property of `this` object	No
	nsIObserverService	Exported property	No
	nsIScriptableInputStream, nsIBinaryInputStream	Return value of exported function	No
	nsISocketTransportService, nsISocketTransport	Property of `this` object	No
	nsIInputStreamPump	Property of `this` object	No
	Instance of the imported `Socket` module	Property of `this` object	No
Most Recent Tab	nsIPrefBranch	Property of `this` object	No
	window	Function return	No
Open Web Apps	nsIPrefBranch, window	Property of `this` object	No
	Reference to built-in SQLite database	Property of `this` object	No
	nsIObserverService	Exported property	No
Recall Monkey	nsIIOService, nsIFaviconService	Property of `this` object	No

Table 8: Capability leaks in Jetpack addons.

Capability leaks in core modules: Beacon discovered two kinds of capability leaks in the core modules. First, capability leaks that occur due to the intended functionality of the module and must therefore be white-listed. Second, capability leaks that occur due to exporting direct references to privileged objects. We list two examples which are representative of the nature of capability leaks in the core modules.

- **window-utils**: The core module `window-utils` as part of its intended functionality exports utility methods to access and track the browser's windows. As mentioned in Section 2, the Jetpack framework executes each module within a sandbox without access to the privileged `window`, `document` or `gBrowser` objects. On analyzing `window-utils`, Beacon reported several capability leaks for methods and properties defined on the `exports` interface that return references to the `window` and `document` objects. Since all of these violations were due to intended functionality as documented in the Jetpack addon SDK [7], we white-listed the offending leaks for the `window-utils` module.
- **xpcom**: The `xpcom` module provides functionality to register a user-defined component with XPCOM and make it available to all XPCOM clients. This module also exposes the `XPCOMUtils` module which offers several utility routines for the components loaded by the JavaScript component loader. Due to the privileged nature of these utility routines, we modeled the `XPCOMUtils` module as a capability source. Our analysis of the `xpcom` module reported a capability leak which we confirmed manually as the reference to the exported `XPCOMUtils` module.
  Exporting a reference to a privileged interface is inconsistent with the philosophy of Jetpack. We believe that instead of the reference to the `XPCOMUtils` module,

separate accessor methods that invoke its functionality should be exported by the `xpcom` module. We reported our observation about the `xpcom` module to Mozilla and they agree with our suggestion to wrap the functionality of `XPCOMUtils` with `xpcom` accessors to decrease the surface area for vulnerabilities.

Capability leaks in Jetpack addons: Capability leaks discovered by Beacon in the Jetpack addons can be classified into four categories. The first category of leaks occurs due to export of capabilities through direct references of privileged objects or due to function objects which return capabilities on invocation. The second class of leaks occurs when a module attaches a capability to an exported function's `this` object. The third class of capability leaks occur if the module utilizes the functionality of a core module which itself leaks capabilities, such as `window-utils` or `xpcom`. Lastly, we also observed capability leaks when a Jetpack addon uses third-party modules which themselves leak capabilities. We describe two popular Jetpack addons which demonstrate all four classes of capability leaks.

- **Customizable Shortcuts**: Customizable Shortcuts is a popular Jetpack addon with over 5000 users. It enables users to easily create keyboard shortcuts to customize the Web browser. We analyzed the addon using Beacon and found 3 capability leaks which cover three out of the four classes of leaks. The first leak results from one of the modules exposing a method that on invocation returns reference to the entire preferences tree, instead of the sub-tree specific to the addon. Accessing the entire preferences tree is not recommended since tree modifications on other branches could result in inadvertent loss of user data.
  The second capability leak occurs in a module which exports a wrapper method over the `window-utils` core module. The wrapper invokes functions on `window-utils` which return references to the `window` and `document` objects.
  The last capability leak occurs as a result of the module attaching an instance of the `nsIAtomService` XPCOM interface to the exported function's `this` object. Although, the `nsIAtomService` interface does not provide any security critical functionality, leaking capabilities implicitly through the `this` object is a bad programming practice.
  On manually verifying the leaks, we observed that none of the leaked capabilities was being used by other modules in the Jetpack addon. This suggests that the module author inadvertently exported the capability instead of keeping it local to the module.
- **Firefox share**: Firefox share is a Jetpack addon by Mozilla Labs which allows fast and easy sharing of links from any Web page. This addon has 25 modules with over 5300 lines of JavaScript code. Several of these modules have been reused from another Jetpack addon, Open Web Apps, also by Mozilla Labs.
  Analyzing Firefox share with Beacon, we discovered 10 diverse capability leaks ranging from leaking preference trees, the `window` object, access to a built-in SQLite database to leaking socket services, which would enable a module to leverage benefits equivalent of using raw UDP/TCP sockets. Table 8 enumerates all the observed violations in Firefox share. On manual verification, we observed that in each case the leaked capability was never invoked from any another module. This clearly indicates that the leaks were inadvertent.

We also found that four of the leaks originated in the code modules that were shared with Open Web Apps. This demonstrates that sharing of over-privileged code modules exacerbates capability leaks.

5.1.1 Accuracy: Beacon detected a total of 36 capability leaks in over 600 modules. For each capability leak, we manually validated the results and observed *no* false positives. However, Beacon could miss capability flows due to a combination of the following reasons:

- **Dynamic features**: Our analysis currently does not handle some of the dynamic and reflective features available in JavaScript. For example, privilege propagation through iterators, generators and reflective constructs like `arguments.callee` are not modeled. Accurate propagation of privileges for such constructs cannot be achieved statically alone and requires dynamic analysis [20, 21].
- **Unsupported constructs**: There are a few constructs in JavaScript for which the WALA analysis engine throws exceptions, and thus they are not supported by Beacon. Such constructs include `for..each`, `yield` and `case` statement over a variable. We re-wrote all instances of such constructs (by hand) in the Jetpack modules to make them amenable to analysis. Although hard to quantify, it is possible that the re-written code may miss some capability flows.
- **Unmodeled constructs**: There are some constructs which have not been appropriately modeled yet in our analysis. These include nested `try/catch/throw` sequences, `eval` and `with`. During our experiments, we found ***no*** instance of either `eval` or `with` in any of the modules.
  Also, our analysis currently does not model DOM function calls, like `setAttribute` and property assignments, like `innerHTML`. Such constructs are handled similar to normal JavaScript function calls and property assignments and could affect capability flows.
  Although foreign function calls, like those invoked on imported modules, are modeled, the analysis does not consider the taint value of arguments passed to them. Instead, the analysis determines the taint value of function returns by consulting the module's summary. Ignoring taint values of arguments of foreign functions could also affect the detection of capability flow.
- **Latent bugs**: Lastly, in-spite of exhaustive testing, it is possible that there are latent bugs in Beacon or the automated module summary generation which might affect capability flows.

5.2 Capability use

The Jetpack framework automatically generates a manifest for each Jetpack addon that provides a dossier about the core modules 'required' by the addon, but provides no information about the XPCOM interfaces invoked by the modules in the addon. As revealed in Section 5.1, a large number of capability leaks originated from the direct use of XPCOM interfaces. In this section, we analyze the Jetpack addons and determine the XPCOM-level capabilities associated with them. A concrete understanding of the capabilities associated with a Jetpack addon is useful to both the end-user and Mozilla itself.

XPCOM Interface	# Jetpack addons	XPCOM Interface	# Jetpack addons
nsIWindowMediator	18	nsIWindowWatcher	4
nsIIOService	10	nsIFaviconService	4
Services	8	AddonManager	3
nsIPrefService	6	nsILocalFile	3
nsIProperties	5	nsIObserverService	3

Table 9: Top 10 XPCOM interfaces used in Jetpack addons.

- Addon reviewers at Mozilla can use capability leak analysis to publish fine-grained Jetpack addon manifests that accurately lists all its capabilities. This would be helpful to end-users in making a well-informed choice when installing an addon. For example, if a Jetpack addon invokes the `nsICookieManager` and also has access to the network, then the end-user can be made aware of the fact that the addon is capable of reading user cookies from all domains and sending them over the network.
- A capability analysis of existing Jetpack addons would help Mozilla in two ways. First, the analysis would identify the set of XPCOM interfaces that are most widely used by developers and for which there do not exist any core modules. This knowledge would help Mozilla in prioritizing the development of core modules. Secondly, the analysis would help the curators at Mozilla to identify addons that use XPCOM interfaces for which a core module already exists. The curator can then suggest the desired modifications to the developer and ensure that all Jetpack addons conform to the hierarchical model where the developer maximizes the use of the built-in core modules for the Jetpack addon functionality.

Core module	# Jetpack addons	Core module	# Jetpack addons
self	243	request	101
tabs	160	chrome	94
widget	157	panel	83
page-mod	126	simple-storage	82
context-menu	117	selection	52

Table 10: Top 10 core modules used in Jetpack addons.

To understand the usage pattern of capabilities in Jetpack addons, we modify Beacon to collect two kinds of capability usage characteristics. First, we track all heap object creations that occur when a Jetpack addon invokes an XPCOM interface. Second, we measure the usage of core modules, *i.e.,*the number of core modules imported using a `require` call.

Figure 5 shows the frequency distribution of XPCOM interfaces for the 359 Jetpack addons which directly invoke atleast one XPCOM interface. We observe that 46 of the addons directly invoke XPCOM functionalities, with one Jetpack addon (Firefox share by Mozilla Labs) invoking 14 XPCOM interfaces. Thus over 12% of Jetpack addons directly use XPCOM to include functionality and features not available in the core modules. We believe that as the Jetpack framework becomes popular, this number will increase and along with it the number of modules that leak capabilities.

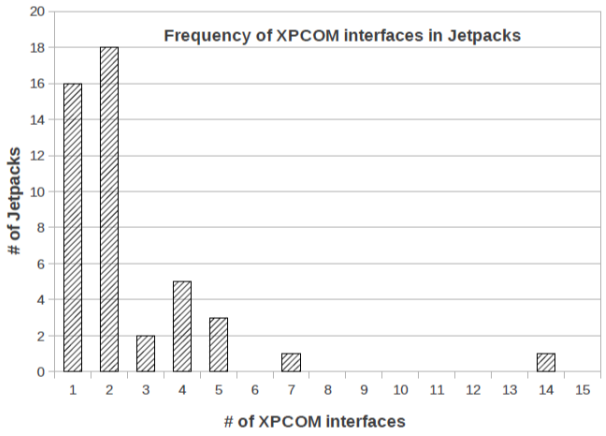

Fig. 5: Frequency of XPCOM interfaces used in Jetpack addons.

Tables 9 and 10 list the top 10 XPCOM interfaces and core modules currently in use by Jetpack addons. We observe that 5 of the XPCOM interfaces listed in Table 9, namely `nsIWindowMediator`, `nsIPrefService`, `nsIWindowWatcher`, `nsILocalFile` and `nsIObserverService`, are used by addon authors even though there exist core modules that provide equivalent functionality. For example, the core module `preference-services` provides functionality equivalent to the XPCOM interface `nsIPrefService`. Two of the popular interfaces `nsIIOService` and `Services` provide rich functionality that currently do not have any functionally equivalent core modules. Although a Jetpack addon author can access these capabilities by requesting `chrome` privileges, it increases the privileges associated with the module manifold. The surface area for vulnerabilities in Jetpack addons would greatly reduce if Mozilla could provide core modules for privileged, but frequently used XPCOM interfaces.

Careless handling of multiple capabilities in a module could result in capability leak through the module's `exports` interface. To determine if the modules in Jetpack addons can be split up into better-confined subsets of authority, we used Beacon to detect all modules which accessed more that one XPCOM interface. We grouped the XPCOM interfaces by their functionality and identified modules that used XPCOM interfaces from different categories. If a module uses functionalities from more than one category, then it is a candidate for isolating the authorities used by the module.

We grouped the XPCOM interfaces into 6 categories — namely Application, Browser, DOM, I/O, Security and Miscellaneous — each representing distinct classes of functionalities, All XPCOM interfaces that access application or user preferences, create application threads, etc. are categorized under Application. The category Browser contains interfaces that represent browser neutral functionality like access to timers and console. DOM provides access to the `window` and `document` objects. Services that handle browser permissions and cookies are grouped under Security, while interfaces which

Jetpack addon	Module name	Categories					
		Application	Browser	DOM	I/O	Security	Misc.
Add-on Builder Helper	main	✓	✓				
	bootstrap	✓			✓		✓
Auto Shutdown NG	countdown	✓		✓	✓		✓
Awesome Screenshot	ui			✓	✓		
Bookmarks Deiconizer	main	✓	✓	✓	✓	✓	✓
Browser Sign In	sessions		✓				
Do Not Fool	localization				✓		✓
Fastest Search	main	✓		✓	✓		✓
Firefox Share	api				✓		✓
	oauthconsumer	✓		✓			
	socket	✓			✓		
	typed-storage				✓		✓
Image2Icon	main		✓	✓			
LepraPanel 2	main		✓	✓	✓	✓	
Memory Meter	main	✓	✓	✓	✓	✓	✓
Open Web Apps	api				✓		✓
	oauthconsumer	✓		✓			
	typed-storage				✓		✓
PriceBlink	main	✓	✓				
Read Later Fast	main		✓	✓			
Recall Monkey	helper			✓	✓		
	main	✓	✓	✓	✓	✓	✓
Snaporama	main	✓	✓	✓	✓	✓	✓
Springpad	main			✓	✓	✓	
Socat	main	✓	✓		✓		✓
Wsad.it Bookmarks	main				✓	✓	

Table 11: List of Jetpack modules accessing multiple categories of XPCOM interfaces.

require access to the network, file system or storage come under I/O. The remaining interfaces are grouped as Miscellaneous.

We found 26 modules in 19 Jetpack addons, where each module invoked XPCOM interfaces to obtain capabilities of different nature. Table 11 lists the findings. We observe that these modules request a wide variety of authorities, with 4 modules requesting access to all 6 categories. We believe that such modules could be split into better-confined subsets.

5.2.1 Accuracy: We evaluated the accuracy of capability use analysis by comparing the results against the ground truth. By manually analyzing all the modules, we found 53 Jetpack addons which had direct invocations to XPCOM interfaces. Beacon detected 46 addons with XPCOM capabilities. The remaining 7 addons invoked XPCOM interfaces from within event handling code (which Beacon does not model — for reasons stated in 3.1).

5.3 Over-privileged modules

The Jetpack addon documentation outlines several guidelines about best practices for developing modules. One of them recommends module authors to follow the principle of least authority (POLA) [8]. To study how the existing core modules conform to this guideline, we analyzed all 77 core modules using Beacon. Our analysis revealed 10 over-privileged core modules.

Core module	Privilege	Severity
`file`	Directory service	Moderate
`hidden-frame`	Timer	None
`tab-browser`	Errors	None
`content/content-proxy`	Chrome	Critical
`content/loader`	File	Moderate
`content/worker`	Chrome	Critical
`keyboard/utils`	Chrome	Critical
`clipboard`	Errors	None
`widget`	Chrome	Critical
`windows`	XPCOM, apiUtils	Critical

Table 12: List of core modules violating POLA.

Table 12 lists the core modules and the nature of the unused privilege. We observe 11 instances of additional privileges which are requested but never utilized in the module code. We also see that 5 of the core modules request critical capabilities like `chrome` and `XPCOM` but never use it. Two modules request `file` and `directory-service` capabilities, which give them privileges to navigate through and read/write to the file system, while the remaining three modules import harmless capabilities which are never used. We contacted Mozilla and notified them about the over-privileged core modules, which they acknowledged as refactoring oversights [6].

5.3.1 Accuracy: To measure the accuracy of false positives in detection of over-privileged modules, we manually validated the Beacon's results for all 77 core modules. Beacon generated a total of 18 warnings for all core modules, out of which 11 were true positives, while the remaining 7 were false positives. On verifying the 7 instances of false positives, we observed that the over-privileged objects were defined in the module's global scope but were used within event handling code. As mentioned in Section 3.1, Beacon does not analyze event handling code, thereby causing false positives.

6 Related Work

Recently, there has been much interest in the analysis of browser extensions for security. To our knowledge, this paper is the first to analyze the Jetpack addon framework.

Sabre [20] and Djeric and Goel [21] both present dynamic information-flow tracking system to detect insecure flow patterns in JavaScript extensions. While the goal of these systems is to detect extensions that can leak sensitive browser data, Beacon instead aims to detect poor software engineering practices in Jetpack modules and addons that can potentially lead to such situations. Moreover, Beacon employs static analysis, which makes it better suited to proactively prevent unwanted information flows in browser extensions.

VEX [13, 14] also implements static analysis of JavaScript to study vulnerabilities in extensions. It implements a flow- and context-sensitive analysis that was applied to over 2400 Firefox addons to detect unsafe programming practices. In VEX, vulnerabilities are specified as bad flow patterns; the analysis attempts to verify the absence

of these patterns in addons. While VEX was originally applied to traditional Firefox addons, it can also be applied to Jetpack modules to detect bad programming patterns. Beacon's analysis goes further to detect capability leaks that may violate modularity, and violations of POLA, which VEX cannot. Unlike VEX, Beacon employs flow- and context-insensitive analysis of JavaScript. Despite the use of lower-precision analysis, Beacon is able to find real vulnerabilities in Jetpack modules and addons.

IBEX [24] provides tools for extension curators to detect policy violating JavaScript extensions. However, IBEX is a framework for specifying fine-grained access control policies guarding the behavior of monolithic browser extensions, while Beacon performs information-flow for modular JavaScript extensions and is designed to detect modules that violate POLA or leak capabilities across module interface. IBEX also requires extensions to first be written in a dependently-typed language (to make them amenable to verification), following which they are translated to JavaScript. In contrast, Beacon works directly with Jetpack extensions written in JavaScript.

More generally, there has been much recent work on static analysis of JavaScript code executing on Web pages. Beacon borrows and builds upon the techniques introduced in these papers (discussed below), but applies them to the analysis of the Jetpack framework.

The core analysis of Beacon is most similar to that of Gatekeeper [22]. While Gatekeeper was originally applied to study the security of small JavaScript-based widgets, we applied Beacon to study capability leaks in Jetpack addon. Actarus [23] is another static analysis based system that studies insecure flows in JavaScript Web applications. Its set of sources and sinks are thus based on rules targeting specific vulnerabilities. For example, the DOM property `innerHTML` or the method `document.write` is a sink because they facilitate code injection attacks. Beacon in comparison targets Jetpack addon, which have well defined sources (`require` and XPCOM) and sinks (`exports`) for each module. ENCAP [28] is related to Beacon in the domain of identifying capability leaks via static analysis. Like Beacon, ENCAP implements a flow- and context-insensitive static analysis of JavaScript, but Beacon differs in both its implementation and application domain. ENCAP uses static analysis to detect API circumvention, where as Beacon detects capability flows in modular JavaScript code.

Chugh *et al.*present staged information flow [18], an analysis infrastructure for JavaScript code. The goal of their original analysis was to detect insecure flows in JavaScript Web applications. However, they developed a novel phased analysis that would allow new code generated in previous phases to be analyzed. Beacon can possibly use these techniques to analyze dynamic constructs, such as `eval` and `with`.

Although not directly related to the analysis of the Jetpack framework, Google Chrome's extension architecture also encourages a modular design [15]. Its extensions consist of a scriptable part, and a native part, and each extension is required to specify its resource requirements upfront in a manifest. The contents of the manifest are then enforced by the browser, thereby limiting the effect of any exploits against the extension. However, recent works have shown that this model may be insufficient to ensure the security of Chrome extensions [17, 25].

7 Conclusions

In this paper, we described Beacon, a system for capability flow analysis of JavaScript modules. Beacon uses static analysis to detect flow of capabilities through the module's `exports` interface. The techniques used by Beacon are generic, and can detect capability leaks in any modular JavaScript code base, e.g., node.js [9], Harmony modules [4], SproutCore [11]. However, our focus was on browser addons implemented using Jetpack. Beacon cannot directly be applied to non-modular addons.

We implemented Beacon and used it to analyze 77 core modules from Mozilla's Jetpack framework and another 359 Jetpack addons. In total, Beacon analyzed over 600 Jetpack modules and detected 12 capability leaks in 4 core modules and another 24 capability leaks in 7 Jetpack addons. Beacon also detected 10 over-privileged core modules. We have shared the details with Mozilla who have acknowledged our findings for the core modules.

In conclusion, the Jetpack framework attempts to improve how scriptable extensions for the Mozilla Firefox browser are developed. Although it provides guidelines for developing modular addons and recommends POLA, it does not enforce these guidelines. Our evaluation of the Jetpack framework suggests that even heavily-tested core modules may contain capability leaks. The use of a tool such as Beacon during addon development can help prevent such leaks.

The overall security of the Jetpack framework can further be improved by dynamically enforcing permissions requested in extension manifests and by deep freezing the `exports` object. Dynamic enforcement of manifests will ensure that addons are not able to access any resources that they have not explicitly requested. Deep freezing the `exports` object will prevent any capability leak through event handlers. We are investigating other design recommendations in current work.

Acknowledgments: We thank Myk Melez, Brian Warner, David Herman and the whole Jetpack team at Mozilla for helping us in better understanding of the framework. This work was supported in part by NSF grants CNS-0952128 and CNS-0915394.

Bibliography

[1] Customizable shortcuts. https://addons.mozilla.org/en-US/firefox/addon/customizable-shortcuts/L.
[2] Firebug: Web development evolved. http://getfirebug.com.
[3] Greasespot: The weblog about Greasemonkey. http://www.greasespot.net.
[4] Harmony modules. http://wiki.ecmascript.org/doku.php?id=harmony:modules.
[5] Jetpack. https://wiki.mozilla.org/Jetpack.
[6] Jetpack addon refactoring oversights. https://github.com/mozilla/addon-sdk/pull/291.
[7] Jetpack sdk. https://addons.mozilla.org/en-US/developers/docs/sdk/1.3/.
[8] Jetpack security model. http://people.mozilla.com/~bwarner/jetpack/components.
[9] node.js. https://nodejs.org.

[10] NoScript—JavaScript blocker for a safer Firefox experience. http://noscript.net.
[11] Sproutcore. http://sproutcore.com/.
[12] Xul. https://developer.mozilla.org/En/XUL.
[13] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *Usenix Security*, 2010.
[14] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vetting browser extensions for security vulnerabilities with VEX. *CACM*, 54(9), September 2011.
[15] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*, 2010.
[16] Rafael Caballero-Roldn, Yolanda Garca-Ruiz, and Fernando Senz-Prez. Datalog educational system. www.fdi.ucm.es/profesor/fernan/des/.
[17] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *UC Berkeley Technical Report*, 2012.
[18] R. Chugh, J. Meister, R. Jhala, and S. Lerner. Staged information flow in JavaScript. In *ACM SIGPLAN PLDI*, 2009.
[19] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.
[20] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript based browser extensions. In *ACSAC*, 2009.
[21] Vladan Djeric and Ashvin Goel. Securing script-based extensibility inweb browsers. In *Usenix Security*, 2010.
[22] S. Guarnieri and B. Livshits. GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security*, 2009.
[23] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *ISSTA*, 2011.
[24] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *IEEE S&P*, 2011.
[25] Guanhua Yan Lei Liu, Xinwen Zhang and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *NDSS*, 2012.
[26] Mozilla Developer Network. Xpcom. developer.mozilla.org/en/XPCOM.
[27] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
[28] Ankur Taly, Ulfar Erlingsson, Mark S. Miller, John C. Mitchell, and Jasvir Nagra. Automated analysis of security-critical javascript apis. In *IEEE S&P*, 2011.
[29] IBM Watson. Watson libraries for analysis. wala.sourceforge.net/wiki/index.php/Main_Page.
[30] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2004.

All URLs were last accessed on March 21, 2012.