Published in Proceedings of ECOOP'14, the 28th European Conference on Object-Oriented Programming

Retargetting Legacy Browser Extensions to Modern Extension Frameworks

Rezwana Karim[1], Mohan Dhawan[2], and Vinod Ganapathy[1]

[1] Rutgers University, Piscataway NJ, USA
{rkarim,vinodg}@cs.rutgers.edu
[2] IBM Research, New Delhi, India
mohan.dhawan@in.ibm.com

Abstract. Most modern Web browsers export a rich API allowing third-party extensions to access privileged browser objects that can also be misused by attacks directed against vulnerable ones. Web browser vendors have therefore recently developed new extension frameworks aimed at better isolating extensions while still allowing access to privileged browser state. For instance Google Chrome extension architecture and Mozilla's Jetpack extension framework.

We present Morpheus, a tool to port legacy browser extensions to these new frameworks. Specifically, Morpheus targets legacy extensions for the Mozilla Firefox browser, and ports them to the Jetpack framework. We describe the key techniques used by Morpheus to analyze and transform legacy extensions so that they conform to the constraints imposed by Jetpack and simplify runtime policy enforcement. Finally, we present an experimental evaluation of Morpheus by applying it to port 52 legacy Firefox extensions to the Jetpack framework.

Keywords: JavaScript browser extensions. Privilege separation.

1 Introduction

Extensions enhance the core functionality of Web browsers, enabling end users to customize the look and feel of their browsing experience. The ease with which browser extensions can be written, downloaded and installed and the features that they enable have all contributed tremendously to their popularity, as well as to the browsers that they target. Browsers such as Mozilla Firefox and Google Chrome have galleries with thousands of extensions implementing a wide array of features. Popular extensions often have in excess of a million users.

To support extensions, browsers typically expose an API that gives access to privileged browser objects. For example, Mozilla's XPCOM (cross-domain component object model) API [25] allows browser extensions to access the file system, the network, the cookie store, and user preferences, among others. Such a rich API is often necessary to implement extensions with useful features. In sharp contrast, code that executes within a Web page is often tightly sandboxed by the browser, *e.g.,* using the same-origin policy, and does not have access to such privileged browser APIs.

Unfortunately, browser extensions do not undergo the same quality control as the rest of the browser, and are riddled with vulnerabilities. In a recent study of over 2400

R. Jones (Ed.): ECOOP 2014, LNCS 8586, pp. 463–488, 2014.

Mozilla Firefox extensions, Bhandakavi *et al.* [8] found several instances of insecure programming practices that can easily be exploited for malicious purposes. Any such exploit would endow the attacker with access to privileged browser APIs, thereby completely undermining the security of the Web browser.

Given such concerns, browser vendors have begun to develop new frameworks that aim to better isolate extensions [9, 2, 6, 5]. These frameworks force extension authors to adhere to core security principles, such as privilege separation and least privilege to some extent. They partition extensions to limit how extensions access privileged browsed objects. An attacker who hijacks one of the partitions of such an extension is unable to access privileged browser objects available to other partitions. Mozilla's Jetpack framework and the Google Chrome extension model are two popular examples of modern extension frameworks that use these techniques to improve extension security.

While the quantitative impact of such frameworks at reducing attacks against extensions is as yet unknown, it is qualitatively clear that by embracing first principles, they improve extension security. However, such frameworks require extensions to be written from ground up, adhering to the programming disciplines that they enforce. To be applicable to legacy extensions, the extensions must be ported to the new frameworks. However, doing so manually would be expensive and time-consuming.

In this paper, we present Morpheus, a static analysis and transformation tool that allows legacy extensions to be systematically ported into modern extension frameworks in a manner that allows enforcement of fine grained security policies without any modification to browser runtime. Our prototype targets legacy Mozilla Firefox extensions, and rewrites them to make them compatible to the Jetpack framework while conforming to the security principles. We chose to focus on Firefox because of the abundance of legacy extensions for this browser. There are currently over 9000 extensions available for Firefox. Morpheus targets an important subset of these extensions, those written fully in JavaScript. Rather than require these extensions to be rewritten for Jetpack from scratch, Morpheus preserves the investment in these extensions and provides a path for automatically refactoring them to work in Jetpack. We have applied Morpheus to port 52 popular Firefox extensions into the Jetpack framework, and are actively applying it to more extensions from the Firefox extension gallery.

This paper makes the following contributions:

- We identify the key challenges in building a reliable and usable toolchain (Morpheus) for systematic conversion of legacy Firefox extensions to the more secure Jetpack framework.
- We present an automated transformation toolchain to partition legacy extension code into Jetpack modules that satisfy the principle of least privilege. Each module encapsulates objects corresponding to sensitive browser APIs and enables accessor methods which provide the required API functionality.
- We present a policy checker framework for Jetpack extensions. The modular and extensible architecture of Jetpack extensions allows developers to seamlessly add or remove security policies without affecting the rest of the code.
- Our evaluation with a suite of 52 popular legacy extensions demonstrates that the design of Morpheus is practical and it is deployable for real world use.

2 Overview

In this section, we describe the architecture of legacy extensions, with a particular focus on issues that motivated browser vendors to develop new extension frameworks. We then discuss the key components of the new Jetpack framework from Mozilla.

2.1 Threats to Extension Security

Browser extensions are written using open technologies such as HTML, CSS and JavaScript, but they often utilize privileged browser APIs to perform useful tasks. For example, Mozilla's XPCOM API gives an extension access to the file system, the network, and sensitive browser state such as cookies and browsing history. The goal of an attacker is to misuse the extension to access the capabilities provided by browser APIs.

A typical browser extension can interact with content on Web pages and any remote server on the Internet. For example, a DisplayWeather extension may access the Web page to search for locations in the text as specified by the user, and its home server to get the corresponding weather data to be shown in the Web page itself. An attacker can hijack an extension by either (1) tricking the user into visiting a malicious Website and then exploiting vulnerabilities in the extension, or (2) compromising the extension's communication with its home server, *i.e.,* the attacker can inject malicious packets in the network stream or compromise the remote server to which the extension communicates.

Browsers attempt to safeguard against the first class of attacks by isolating the execution of JavaScript code on the Web page (unprivileged ***content scripts***) from the JavaScript code executing within the extension (privileged ***chrome scripts***). This isolation of content scripts from chrome scripts limits the threats posed by a Web attacker by disallowing direct access to sensitive browser APIs. Nevertheless, there are often bugs in this isolation mechanism, leading to exploits. To defend against the second class of network-based attacks, extensions can use SSL to secure their connection with their home server.

2.2 Legacy Extensions on Firefox

Consider Figure 1, which shows a snippet from the DisplayWeather extension that we developed. The extension provides options to overlay weather information on a browser panel for which it reads the zipcode from persistent storage. In lines 1-6, the function `getZipCode` reads the file `'zip.txt'` from the user's profile directory to retrieve the zipcode for the user specified location. In line 2, `import` attaches the `FileUtils` object to the extension's global namespace. `FileUtils.jsm` internally invokes XPCOM APIs to enable all file I/O operations. Lines 9-28 define the `Weather` object that encapsulates properties and methods to fetch weather data from a remote server. The method `requestDataFromServer` defined in lines 16-27 uses `XMLHttpRequest` to fetch weather data for a given zipcode from a remote server. Line 30 registers a `click` event listener with the extension's icon in the browser's status bar to display weather in a panel. In lines 33-37, the code creates an event listener `addWeatherToWebpage` to overlay weather information on the Web page, whenever a new Web page is loaded.

```
(1)  function getZipCode(locationStr){
(2)     Components.utils.import('resource://gre/modules/FileUtils.jsm');
(3)     var dir = 'ProfD', filename = 'zip.txt';//get the 'zip.txt' file from profile directory
(4)     var file = FileUtils.getFile(dir, [filename]);
(5)     var locationZipcodeMap = readFile(file);
(6)     return locationZipcodeMap[locationStr]; //retrieve zipcode for the location
(7)  }
(8)  ...
(9)  var Weather = {
(10)    temperature: null,
(11)    ...
(12)    getWeatherData: function(zipcode){
(13)       Weather.requestDataFromServer(zipcode);
(14)       return processWeatherData(Weather.temperature);// format weather data
(15)    },
(16)    requestDataFromServer: function(sendData){
(17)       var httpRequest = new window.XMLHtttpRequest();
(18)       ...
(19)       //set the listener to handle response from Server
(20)       httpRequest.onreadystatechange = function(){
(21)           // extract temperature data from response and set Weather.temperature
(22)          Weather.extractTemperature(httpRequest.response);
(23)             ...
(24)       }
(25)       httpRequest.open('GET', serverUrl, true);
(26)       httpRequest.send(sendData);//contact remote server
(27)    }
(28) }
(29) //Add the click listener to the extension's icon to show Weather in panel
(30) document.getElementById('weatherStatusBar').addEventListener
              ('click', showWeatherInPanel, false);
(31) ...
(32) window.addEventListener('DOMContentLoaded', addWeatherToWebpage, false);
(33) function addWeatherToWebpage(){
(34)    var locationStr = getLocationFromWebpage(gBrowser.contentDocument);
(35)    var temperature = Weather.getWeatherData(getZipCode(locationStr));
(36)    modifyWebpageContent(gBrowser.contentDocument, temperature);
(37) }
```

Fig. 1. Code snippet from the DisplayWeather extension

Lines 34-36 identify all DOM[1] elements that contain a user-specified location in the active Web page and invoke `getWeatherData` method defined on the `Weather` object to retrieve latest weather updates. The method `modifyWebpageContent` in line 36 actually overlays the weather information on the active Web page.

This example highlights several features used by legacy Firefox extensions:

(1) *Unified JavaScript heap*: Mozilla's legacy extension development environment provides a unified heap for all JavaScript code execution. Both privileged chrome scripts and unprivileged content scripts reside in the same heap, raising the risk of shared references. For example, line 36 invokes the `modifyWebpageContent` method with a reference to the document object of the active Web page. Mozilla uses XrayWrappers (also know as XPCNativeWrappers) to isolate the untrusted references of the content JavaScript from the chrome JavaScript. However, this mechanism has a history of exploitable bugs [9, 29]. If this interface is exploited, and the user navigates to a malicious Web page, the `document` object would belong to the attacker, who could then influence the execution of the privileged code within the extension [30].

A second consequence of having a unified heap for JavaScript execution results is that top-level objects declared in chrome scripts are attached as properties of the global object. This often results in namespace collisions across different extensions or even

[1] Document Object Model (DOM) provides a structural representation of the document, enabling developers to modify its content and appearance using JavaScript.

different chrome scripts within the same extension. Further, since globals defined in one script can be accessed and modified from another script, data races may occur.

(2) *Privileged objects*: All chrome scripts have default access to the global `window` object and its properties. The `Components` object is a special property of the `window` which provides access to the browser's sensitive XPCOM APIs. If an attacker gets a reference to the `Components` object, he effectively has control over the entire browser. The fact that the `Components` object is so powerful and is yet available to all scripts by default is a significant threat to security in a shared heap environment.

(3) *Chrome DOM*: Much as the DOM API available to content scripts on a Web page, chrome scripts also have access to the chrome DOM. The chrome DOM is responsible for the visual representation of the browser's UI including toolbars, menus, statusbar and icons. Since much of Firefox's UI is also written in JavaScript, chrome scripts can programmatically access and modify the browser's entire UI (line 30).

The issues discussed above stem in part due to the architecture of Mozilla's legacy extension framework. Parts of the browser itself are written in JavaScript, as are extensions. With a unified heap and lack of any isolation primitives in the language itself, extension developers must consciously and carefully restrict access to critical functionality. The legacy extension framework makes it easy for developers to commit mistakes, and much prior work has shown the pitfalls of legacy extensions [13, 8, 14, 9].

2.3 The Jetpack Extension Framework

The Jetpack extension framework [2, 20] is an effort by Mozilla to incorporate security principles in the design of the extension architecture, thereby improving the overall security of extensions. Jetpack uses a layered defense architecture to make it harder for an attacker to compromise extensions, and limit the damage done if he succeeds in compromising all or part of the extension. The Jetpack project shares ideological similarities with the Google Chrome extension architecture [9]. It has also been motivated by the goal of easing extension development process with an emphasis on modular development and code sharing, and partly by the new multi-process Firefox architecture [24].

Conceptually, each Jetpack extension has two parts: (1) at least one add-on script (also known as *chrome script*) that interacts with a set of *core modules*, which have access to the sensitive browser APIs, and (2) zero or more *content scripts*. The chrome script(s) execute within the Web browser with restricted but elevated privileges: it must explicitly request access at load time to the browser APIs that it requires access to; any attempt to access other APIs at runtime is blocked. Content scripts interact with the Web page and are unprivileged. In addition, Jetpack incorporates these features:

(1) *Chrome/content heap partitioning.* Chrome and content scripts execute in separate processes. This partitioning guarantees isolation of the JavaScript heap for the chrome and content scripts and prevents inadvertent access by content scripts to privileged references in the chrome code. Communication amongst the chrome and content scripts is made possible through IPC with all messages exchanged in the JSON [3] format.

(2) *Content script integrity.* Content scripts execute in the context of the Web page and a malicious Web page can redefine objects referenced by the content script, thereby

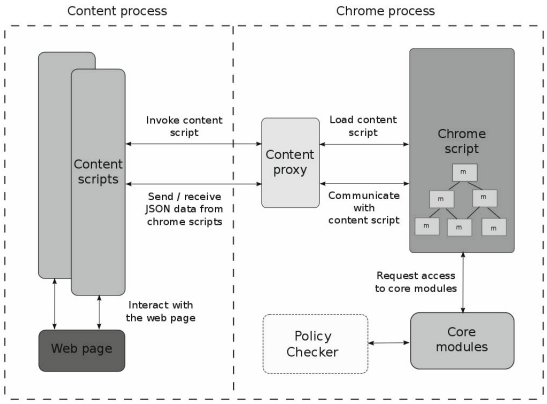

Fig. 2. Architecture of a simple Jetpack extension. Policy Checker is not part of original architecture and is introduced by Morpheus.

affecting its integrity. Jetpack uses *content proxies* to protect the integrity of content scripts. Content proxies allow the content script to access the content on the Web page while still having access to the native objects and APIs (*e.g.,* `document` and `window`), even if the Web page has redefined them.

(3) *Chrome privilege separation.* Jetpack provides developers with a set of core modules that encapsulate the functionality of the privileged browser APIs, thus preventing inadvertent misuse of these APIs by the developer. Further, developers must explicitly request these core modules as required by the extension's chrome scripts. If compromised, this restricts the set of privileges that an attacker can obtain to only those requested by the exploited script.

The Jetpack framework further recommends developers to partition the chrome script and organize an extension as a hierarchy of user modules, each of which may itself request other user modules and zero or more core modules using the `require` interface. The set of privileges thus acquired by each user module is determined statically by analyzing the source code and enforced by the framework at runtime. The Jetpack framework further provides isolation among all modules. Objects declared within a module are local to the module unless exported via the module's `exports` interface.

Figure 2 shows the overall architecture of a Jetpack extension. In summary, Jetpack attempts to improve extension security by separating content scripts from chrome scripts, employing privilege separation for chrome scripts, and restricting the privileges of chrome scripts to those declared at load time. While this architecture does not prevent vulnerabilities in extension scripts, it ensures that the effect of any exploits is contained to the vulnerable components of the extension, and will not give the attacker unbridled access to privileged browser APIs.

However, a compromised chrome script can still trick core modules to access sensitive resources of the attacker's choice. Consider the scenario where the attacker has compromised the chrome script in the DisplayWeather extension, and has changed the parameter value in `FileUtils.getFile()` to read the passwords stored on disk. The core module with privileges to access file-system will then read and return all the saved

passwords to the attacker. Similarly, the attacker can redirect stolen data to an attacker-controlled remote server by changing `serverUrl` in `httpRequest.open()`. In both cases, the attacker does not need to extend the script's privileges at runtime. Instead, lack of policy checker to enforce fine-grained access control enables the attacker to exploit benign extensions even in the security enhanced Jetpack framework.

3 Morpheus

While the Jetpack framework provides clear security benefits to extensions, legacy extensions must be rewritten in Jetpack in order to enjoy these benefits. Morpheus is a static dataflow analysis and transformation tool that automates this process. In this section, we identify the key requirements that Morpheus's analysis and transformation must provide and describe its design.

3.1 Design Requirements

The transformations in Morpheus must perform the following tasks:

(1) *Chrome/content partitioning.* Jetpack requires chrome and content scripts to execute in isolated heaps. Morpheus must analyze the code of the legacy extension and identify object references that should be part of either chrome scripts or content scripts. Code that transitively accesses these object references should also correspondingly be marked for execution within the context of chrome or content scripts.

In Jetpack, chrome scripts interact with content scripts via asynchronous message passing protocols using JSON. In contrast, legacy extensions use synchronous calls for content/chrome communication. For example, calls to `getLocationFromWebPagecontent` and `modifyWebPageContent` (lines 34-36, Figure 1) are synchronous invocations in the legacy extension. Thus, to preserve the control flow of the legacy extension, Morpheus must use the asynchronous communication API available in Jetpack and emulate the synchronous nature of content/chrome communication in legacy extensions.

(2) *Module construction.* The Jetpack framework encapsulates a selection of the privileged browser APIs as core modules and requires developers to arrange their code as user modules to limit the extent of the damage in case of a breach. A Jetpack extension is a hierarchical collection of such core and user modules. Morpheus must identify the use of privileged browser APIs in the legacy extension and create core modules for them. Although creation of user modules is not mandatory, it is recommended. Thus, Morpheus must analyze the legacy extension and extract related functionality that can be compiled into a user module.

Modules interact using the `require` and `exports` interfaces. Although modules are allowed to export privileged objects that they access, doing so would undermine the security of the whole extension (by exposing the object to other modules). Morpheus must therefore ensure that the modules it creates never export references to privileged objects. Instead, they should export accessor methods to these privileged objects, which can be invoked by other modules to achieve their desired tasks. One may argue that exporting

accessor methods is akin to accessing capabilities to achieve the desired functionality. However, as will be described later in Sections 3.3 and 4, isolating capabilities in separate JavaScript modules makes it harder for an attacker to compromise other modules.

(3) *Scope and global objects.* Legacy extensions make frequent use of global objects as shown in Figure 1. Morpheus must ensure that partitioning the code into `chrome/content` and user/core modules does not affect visibility of the globals (or other objects in scope) in the Jetpack extension.

(4) *Policy Checker.* Benign software that exposes an API to third-party code is often vulnerable to the confused deputy problem [16]. To safeguard core Jetpack modules from becoming confused deputies themselves, (see Section 2) and also protect benign-but-buggy extensions, Morpheus must allow enforcement of fine-grained access control and other security policies at runtime. A key requirement here is that the extension code should be oblivious to the security policies and the policy checker implementation.

(5) *Preserve extension UI.* The transformed Jetpack extension must retain the look and feel of the legacy extension. Thus, the browser's UI overlays, including any CSS, XUL and icons, must be appropriately mapped.

In our work to date, we have not attempted to optimize the performance of the transformed extension. The goal of Morpheus is to preserve the investment in legacy extensions, while also improving their security by making them amenable for use within Jetpack. In doing so, Morpheus may degrade the performance of the legacy extension, *e.g.,* by using an asynchronous communication API to emulate synchronous communication. We plan to optimize performance in future work.

3.2 Analyses and Transformations

Morpheus invokes TRANSFORM (see algorithm 1) over the legacy extension to transform it into the corresponding Jetpack extension. TRANSFORM takes in (i) the JavaScript code of the legacy extension L, which has been preprocessed to resolve any global-local scope conflict, (ii) an alias relation A as computed by the CFA2 algorithm [32] over the extension's JavaScript code, and (iii) some basic transformation rules $\mathcal{R}$ (see Table 2). Each transformation rule modifies an expression ξ from the program's abstract syntax tree (AST) T. TRANSFORM in turn invokes algorithms 2(a), 2(b) and 3 to complete the transformation. Table 1 lists the common notations used in all algorithms and rules.

We now discuss in detail the analyses and transformations implemented in Morpheus corresponding to each of the design requirements listed above.

Chrome/Content Separation. To identify object references that must appear in chrome or content scripts, Morpheus identifies the context in which object references and their property accesses should be evaluated. The context of an object reference is the context in which it was declared. Thus, any object declared in chrome code must be evaluated in chrome context and similarly all accesses to content objects must be evaluated in the context of the current Web page (content). For the rest of the paper, we refer chrome context as `chrome` and content context as `content`.

Morpheus uses static dataflow analysis to identify whether code that accesses an object reference should be evaluated in either `chrome` or `content`. Our analysis leverages the dataflow rules given in prior work [32]. The analysis is based on the observation that

Table 1. Common notations used in transformation rules and algorithms

$\mathbb{E}$	Set of all expressions
E_{pa_f}	Fixed property access expression of the form `e.x`, `e['x']`
E_{pa_d}	Dynamic property access expression of the form `e[v]`
E_{pa}	Property access expression where $E_{pa} := E_{pa_f} \cup E_{pa_d}$ where $E_{pa} \subset \mathbb{E}$
E_{mi}	Method invocation expression `e.f( args )`, `e['f']( args )`, `e[vf]( args )`
E_{xpcom}	XPCOM invoke expression, where $E_{xpcom} \subset \mathbb{E}$. It can be one of the two forms, either (i) `Components.classes[.*].getService(Components.interfaces[.*])`, or (ii)`Components.utils.import("resource://gre/modules/*.jsm");`
$E_{objInit}$	Object Literal expression of the form `{ a:1, b:function(){}}`, where $E_{objInit} \subset \mathbb{E}$
E_{decl}	Function/ variable declaration expression, where $E_{decl} \subset \mathbb{E}$. Can be any of the following expressions `const c; let l; var a; var b=5; function foo( ){}`
Expression(η)	Expression for AST node η
Object(ξ)	expression representing object whose property is accessed in expression ξ, where $\xi \in (E_{pa} \cup E_{mi})$
Property(ξ)	expression representing property being accessed in expression ξ, where $\xi \in E_{pa}$
Node(η, ξ)	AST node for expression ξ and a descendant of node η
GetAliasSet($\mathcal{A}, n$)	Consults alias relation $\mathcal{A}$ and returns all may-alias for the node n.
InContent(ξ)	Checks if object denoted by expression ξ belongs to content context.
CanMakeModule($n, \mathcal{T}$)	Decides if code corresponding to AST node n can be extracted and put in a separate module. In our implementation, it embodies the criteria that the object, represented by n, must have at least one method defined as its property that is invoked from outside the object.

Transform($L, \mathcal{A}, \mathcal{R}$)
Input: L : Legacy code, $\mathcal{A}$: alias relation, $\mathcal{R}$: set of rewriting rules
Output: $\mathbb{M}$ a set of Jetpack modules

Initialize:
$\mathcal{T} := AST(L)$; $\mathbb{O} := \emptyset$ `/*Set of AST nodes for object literals*/`
$S :=$ ComputeSensitiveSet($L, \mathcal{A}$) ; $D :=$ ComputeDOMSet($L, \mathcal{A}$)

foreach $n \in$ Nodes($\mathcal{T}$) **do**
  $\xi_n :=$ Expression(n)
  if $\xi_n \in E_{xpcom}$ **then** Rewrite($\xi_n, \mathcal{T}, \mathcal{R}1$) `/*rewrite with require, import core modules*/`;
  else if $\xi_n \in E_{mi} \wedge$ (Node(n, Object(ξ_n)) $\in S \vee$ Node(n, Object(ξ_n)) $\in D$) **then** Rewrite($\xi_n, \mathcal{T}, \mathcal{R}3$) ;
  else if $\xi_n \in E_{pa} \wedge$ (Node(n, Object(ξ_n)) $\in S \vee$ Node(n, Object(ξ_n)) $\in D$) **then** Rewrite($\xi_n, \mathcal{T}, \mathcal{R}2$) ;
  else if $\xi_n \in E_{objInit} \wedge$ CanMakeModule($n, \mathcal{T}$) **then** $\mathbb{O} \cup = \{n\}$;
$\mathbb{M} :=$ ExtractModule($\mathcal{T}, \mathbb{O}$) `/*Creates user modules from the relevant code*/`
return $\mathbb{M}$

Algorithm 1. Transforming legacy extension code to Jetpack modules

JavaScript code in legacy extensions is evaluated in `chrome` unless it specifically makes a transition to access objects in content scripts. There are only a limited number of ways to make a transition from chrome code to content code, *i.e.,* by accessing `content`, `contentWindow` and `contentDocument` properties on selected chrome objects, like `window` and `gBrowser`. This observation forms the basis of our static analysis.

All JavaScript in a legacy extension executes in the same heap, and thus objects have global visibility. To precisely identify which objects must reside in the `chrome` or `content`, Morpheus does a whole program analysis of the legacy extension. It concatenates all JavaScript code within the extension before performing the static analysis. This concatenation includes scripts defined within JavaScript files, event handlers and globals declared within overlay files and also JavaScript code modules. The result of the static analysis is a table where each entry is an object reference and the context in which it should be evaluated.

Static analysis to determine the `chrome`/`content` context of object references can suffer from false positives and negatives when content references are accessed using JavaScript's reflective constructs. This happens, for instance, when object references are used within the `eval` string, or passed as parameters to functions but are accessed

ComputeDOMSet(L, $\mathcal{A}$)
Input: L: Legacy code, $\mathcal{A}$: Alias relation
Output: D : set of AST nodes for DOM objects

Initialize:
$\mathcal{T} := AST(L)$; $D := \emptyset$

foreach $n \in$ Nodes($\mathcal{T}$) **do**
$\xi_n :=$ Expression(n)
$\xi_n^r :=$ rValueExp(ξ_n), $\xi_n^l :=$ lValueExp(ξ_n)
if ($\xi_n^r \in D$)
$\vee(\xi_n^r \in E_{mi} \wedge ((\text{Node}(n, \text{Object}(\xi_n^r)) \in D)$
$\vee$ InContent(Object(ξ_n^r))))
$\vee(\xi_n^r \in E_{pa} \wedge ((\text{Node}(n, \text{Object}(\xi_n^r)) \in D)$
$\vee$ InContent(Object(ξ_n^r))))
$\vee(\xi_n^r \in E_{pa} \wedge ((\text{Node}(n, \text{Object}(\xi_n^r)) \in D)$
$\vee$ InContent(Property(ξ_n^r)))) **then**
$D \cup = \{\text{Node}(n, \xi_n^l)\}$
$A_l :=$ GetAliasSet($\mathcal{A}$, Node(n, ξ_n^l))
$D \cup = A_l$ /*add all alias of ξ_n^l to D*/
return D

(a)

ComputeSensitiveSet(L, $\mathcal{A}$)
Input: L: Legacy code, $\mathcal{A}$: Alias relation
Output: S : set of AST nodes for sensitive objects

Initialize:
$\mathcal{T} := AST(L)$; $S := \emptyset$

foreach $n \in$ Nodes($\mathcal{T}$) **do**
$\xi_n :=$ Expression(n)
if $\xi_n \in E_{xpcom}$
$\vee(\xi_n \in E_{mi} \wedge (\text{Node}(n, \text{Object}(\xi_n)) \in S))$
$\vee(\xi_n \in E_{pa} \wedge (\text{Node}(n, \text{Object}(\xi_n)) \in S))$ **then**
$S \cup = \{n\}$
$A_n :=$ GetAliasSet($\mathcal{A}$, n)
$S \cup = A_n$ /*add all alias of ξ_n to S*/
return S

(b)

Algorithm 2. Algorithms for computation of set of nodes corresponding to (a) content DOM objects and (b) sensitive objects

Table 2. Rewrite rules for expression. Each rule modifies an expression ξ and updates AST T

Rule: $(\xi \Rightarrow \xi') \rightarrow (T \Rightarrow T')$, where $\xi :=$ expression(n). T is set to T' after applying each rule
Rule $\mathcal{R}1$: Import Module $m :=$ get-module-name(ξ) $\xi' :=$ `require('`m`')`
Rule $\mathcal{R}2$**:** Rewrite property access with `setProperty, getProperty` o := object(ξ), prop := property(exp) ($\mathcal{R}2.a$) $\frac{\text{property-read}(T, \xi)}{\xi' :=$ `o.getProperty('p')`$}$ ($\mathcal{R}2.b$) $\frac{\text{property-write}(T, \xi) \quad \text{v} := \text{value-to-store}(T, \xi)}{\xi' :=$ `o.setProperty('p', v)`$}$
Rule $\mathcal{R}3$ **:** Rewrite method invocation with `invoke` o := object(ξ), μ := method(exp), α := arguments(exp) $\xi' :=$ `o.invoke('`μ`', `α`)`
Rule $\mathcal{R}4$**:** Rewrite Global Access with `GlobalGET, GlobalSET` ($\mathcal{R}4.a$) $\frac{\text{Global-read}(T, \xi)}{\xi' :=$ `GlobaGET('`ξ`')`$}$ ($\mathcal{R}4.b$) $\frac{\text{Global-write}(T, \xi) \quad \text{v} := \text{value-to-store}(T, \xi)}{\xi' :=$ `GlobaSET('`ξ`', v)`$}$
Rule $\mathcal{R}5$: Global Write ** This rule creates a new statement $\sigma :=$ `GlobaSET('`ξ`',`ξ`)`

as elements of the `arguments` array within the function. Morpheus currently does not handle such cases and instead relies on the developer to rewrite the code to make it more amenable to analysis, or to manually classify the context of the object reference.

By default, a legacy extension executes in `chrome`, so object references that remain in `chrome` in Jetpack can be evaluated as before. To evaluate objects in `content`, Morpheus considers the `content` as a sensitive resource and models it as a core Jetpack module called `contentDOM`. Algorithm 2(a) identifies all program points corresponding to property accesses of content objects and Morpheus then rewrites these accesses by accessor methods to abstract away the design of the content module from the extension code. For example, the code `gBrowser.contentDocument` in a legacy extension would be rewritten as `gBrowser.getProperty('contentDocument')`. Likewise, the property access `gBrowser.contentDocument.location` would be rewritten as `gBrowse.getProperty('contentDocument').getProperty('location')`.

```
var table = require('core_module_table');
var policyChecker = require('policy_checker');
var _module_ = {
   id: initModule(), /*initializes the module*/
   getProperty: function() {
      var property = arguments[0];
      var violated = policyChecker.check
             (<core module name>, property);
      if(violated){
         return {};
      }
      var ref = table.getReference(this.id);
      switch(property) {
         case '< depends on the core module >':
            var retval = ref[property];
            var newref = < new core module instance>
            table.setReference(newref.id, retval);
            return newref;
         ... /* more case statements */
         default:
            return null;
      }
   },
   /*code for setProperty, invoke*/
}
exports.module = _module_;
```

Fig. 3. Template for secure core module with policy

Morpheus addresses a key challenge that arises as a result of the design of Jetpack's `contentDOM` module. As shown in line 36 in Figure 1, legacy extensions may contain statements that refer to objects in both `chrome` and `content`, *i.e.,* `modifyWebPageContent` is a method defined in the `chrome` while `gBrowser.contentDocument` is the active window's `document` object and is therefore an object in `content`. Moreover, the call to `modifyWebPageContent` is synchronous in the legacy extension. Since the Jetpack framework executes chrome scripts and content scripts in separate processes, they cannot share object references, but only exchange data in JSON format asynchronously. Thus, in the Jetpack counterpart of this extension, the call in line 36 would be asynchronous because `modifyWebPageContent` should be part of content script as they operate on the `gBrowser.contentDocument` from the active Web page. Morpheus addresses this challenge by creating opaque identifiers for objects in the `content` and transmitting these identifiers across the JSON pipe to the `chrome`. Morpheus's transformation also attempts to retain the control flow of the original extension code as intended by the developer (see Section 5).

Module Construction. Modules in Jetpack must ideally not export references to privileged objects. Any such *leaking references* to other modules can lead to privilege escalation attacks, *i.e.,* a module to which a reference is leaked may be able to access a privileged object without explicitly requesting access to it at load time. Morpheus creates extensions that do not export privileged objects. Instead, Morpheus creates module templates (see Figure 3) that export accessor methods to these privileged objects. These modules export only four properties, namely `id`, `getProperty`, `setProperty` and `invoke` to privileged objects. Each module encapsulates a privileged object, which is assigned an opaque identifier (`id`) on module initialization. Other modules access the object using `getProperty` and `setProperty`, which are getter and setter methods, and `invoke`, which allows invocation of methods defined on the privileged object. The first argument to each of `getProperty`, `setProperty` and `invoke` is the property to be accessed followed by a list of arguments. Each of these methods can either return primitive values or an instance of a module. Accessor methods also embody any security policies associated with access to privileged objects. Section 4 discusses the security implications of creating modules in this way.

Morpheus transforms legacy extensions to use core modules designed as above in the following way. It first analyzes the legacy extension to locate the use of browser's privileged XPCOM APIs and generates a list of program points (as shown in algorithm 2(b)) for the property access and methods invoked on corresponding privileged XPCOM

```
ExtractModule(𝒯, 𝕆, 𝒜)
Input: 𝒯 : AST for Legacy, 𝕆 : Set of nodes for object literals, 𝒜 : alias relation
Output: 𝕄 a set of Jetpack modules
Initialize:
  𝕋 := ∅ /*Map from node n ∈ 𝕆 to AST*/ ;  ι := ∅ /*Map from node n ∈ 𝕆 to parentAST from
which it is extracted*/
foreach n_i ∈ 𝕆 do
    T_{n_i} := CopyAstForNode(𝒯, n_i); 𝕋[n_i] := T_{n_i}; ι[n_i] := 𝒯
/*update parent AST for nested object Literal expression*/
foreach n_i ∈ 𝕆 do
    if IsNestedObject(n_i, 𝒯) then
        T^p := FindParentAST(n_i, 𝕋) ; ι[n_i] := T^p /*T^p is the smallest AST T from 𝕋[n_i] such that
        n_i ≠ root(T)*/
foreach n_i ∈ 𝕆 do
    T_{n_i} := 𝕋[n_i] /*AST for node n_i*/
    G_{n_i} := GetGlobalIdentifiers(T_{n_i}) /* G_{n_i} is set of identifiers used but not defined in T_{n_i}*/
    H_{n_i} := GetLocalIdentifiersGloballyUsed(𝒯, 𝕆, T_{n_i}) /*Identifiers defined in T_{n_i} but also used
    in other T*/
    /*H_{n_i} is set of identifiers defined in T_{n_i} and used in other modules*/
    foreach q ∈ Nodes(T_{n_i}) do
        ξ_q := Expression(q)
        if ξ_q ∈ G_{n_i} then Rewrite(ξ_q, T_{n_i}, ℛ4) /*rewrite with GlobalGET, GlobalSET*/;
        else if ξ_q ∈ E_decl then
            σ := CreateNewStatement(lValueExp(ξ_q), ℛ5) /*create a GlobalSET*/
            AddToAst(T_{n_i}, σ)
    m_i := MakeNewModule(T_{n_i}) /*Place the code for AST T_{n_i} in a new module and append
    necessary code*/
    𝕄 ∪ = m_i
    /*modify the parent AST*/
    T^p := ι[n_i] /*get parent AST*/
    ξ_{n_i} := Expression(GetNodeFromAst(n_i, T^p)); Rewrite(ξ_{n_i}, T^p, ℛ1) /* rewrite with require*/
    A_{n_i} := GetAliasSet(𝒜, GetNodeFromAst(n_i, T^p))
    foreach λ ∈ Nodes(T^p) do
        ξ_λ := Expression(λ)
        if ξ_λ ∈ E_mi ∧ (Object(ξ_λ) = ξ_{n_i}) then Rewrite(ξ_λ, T^p, ℛ3) /*rewrite with invoke*/;
        else if ξ_λ ∈ E_pa ∧ (Object(ξ_λ) = ξ_{n_i}) then Rewrite(ξ_λ, T^p, ℛ2) /*rewrite property access*/;
m := MakeNewModule(𝒯) ; 𝕄 ∪ = m /*construct the main module and add to set 𝕄*/
return 𝕄
```

Algorithm 3. Algorithm for extracting user modules

API. Morpheus then rewrites the extension code by replacing all such references as per the rules $\mathcal{R}1$, $\mathcal{R}2$, $\mathcal{R}3$ in Table 2 for the corresponding core module in Jetpack. The Jetpack framework does not provide core modules for all XPCOM APIs, so core modules may have to be supplied separately. We have used our module template to build a suite of core modules for a variety of XPCOM APIs. We developed these core modules by hand, and used an off-the-shelf static analysis tool [17] to verify that these core modules do not export references to privileged objects.

Morpheus also creates user modules by analyzing legacy extension code. The main objective is to partition the chrome script into multiple modules in a way to attenuate the authority of individual modules and limit the effect of a vulnerability exploit. Ideally, user modules should be generated by clustering functions based on access to XPCOM functionality. However objects with privilege to access different XPCOM can be used in a single statement. This makes splitting based on XPCOM access non-trivial, since it would require more precise and sophisticated static analysis and semantic-preserving transformation algorithm. Therefore we adopted a simpler approach of encoding the developer's way of partitioning code.

Morpheus identifies code fragments in the legacy extension that achieve related functionality. The underlying intuition is that these code fragments can then be grouped into a single module. Morpheus uses a simple notion of object ownership to identify related functionality: it identifies a set of functions that are owned by the same object, and groups such functions into a single module. This heuristic is based on the observation that developers often group functionality as object hierarchies that are more likely to access similar, if not the same, XPCOM interfaces within one object. Even though this might provide less meaningful partitions if the developer does not arrange his code using purposeful object hierarchies, our evaluation shows that this approach is practical and we do extract a reasonable number of user modules with most of them accessing only a few core modules. User modules follow the same template as core modules with the difference that the object encapsulated within the module is the one that owns the functions grouped in that module, instead of a sensitive XPCOM object as for core modules. Morpheus rewrites references to the encapsulated objects with a `require` invocation. Algorithm 3 encodes the user modules extraction and rewriting technique.

As shown in line 2 in Figure 1, an extension can load a JavaScript code module (JSM) using an invocation to `Components.utils.import`. The `import` API takes as arguments the URL of the script to be loaded and an optional scope object. On execution of the `import` statement, the array of objects defined in the script (referenced by the URL) is attached to the scope object. In case the scope object is not defined, the imported objects are attached to the global object, *i.e.,* they can be accessed and modified by any script in the extension code. Browser-provided JSMs internally access XPCOM interfaces and therefore are treated as privileged API by Morpheus. Core modules are constructed for them and accesses of such JSMs are rewritten accordingly. In contrast, Morpheus rewrites all JSMs, defined by legacy extension developers, to access only core modules designed as above. However since these JSMs are self contained code fragments with a well defined interface for exporting objects, Morpheus rewrites the entire JSM as a user module, and does not partition it further into smaller modules.

Scope and Global Objects. When Morpheus creates user modules from a legacy extension, it is possible that the resulting user modules may require access to scope or global variables defined in the legacy extension. However, Morpheus creates modules, which are isolated by the Jetpack framework, and therefore cannot share references/updates to scope and global variables. Morpheus therefore creates a new `global` module that (1) stores references to all the scope and global variables, and (2) exports two methods `GlobalGET` and `GlobalSET` to enable access to these variables. It then analyzes all user modules, identifies instances of scope or global variables used (but not defined) and rewrites access to these variables as per rule $\mathcal{R}4$ in Table 2, *i.e.,* using either `GlobalGET` or `GlobalSET`.

Preserving Extension UI. As mentioned in Section 2.2, most of the browser's UI is scriptable, *i.e.,* it can be accessed and modified using JavaScript. Morpheus leverages this ability and generates JavaScript code to dynamically modify the browser's UI on invocation of the Jetpack functionality. To do so, Morpheus analyzes the legacy extension's CSS and XUL overlay files, which represent UI descriptions as XML markups, and dynamically loads the appropriate JavaScript code at runtime to preserve the UI of the legacy extension.

```
(2)  var FileUtils = require('core/FileUtils').module;
(4)  var file = FileUtils.invoke('getFile',dir, [filename]);
(9)  var Weather = require('user/Weather').module;
( )  GlobalSET('Weather', Weather); /*new statement added*/
(30) document.invoke('getElementById','weatherStatusBar')
            .addEventListener('click', showWeatherInPanel, false);
(32) window.invoke('addEventListener','DOMContentLoaded', addWeatherToWebpage, false);
(34) var locationStr = getLocationFromWebpage(gBrowser.getProperty('contentDocument'));
(35) var temperature = Weather.invoke('getWeatherData', getZipCode(locationStr));
(36) modifyWebpageContent(gBrowser.getProperty('contentDocument', temperature));
```

Fig. 4. Code snippet from *Main* module of the transformed DisplayWeather Jetpack extension. Only statements from Figure 1 that are rewritten by Morpheus are shown.

```
(9)  var _module_ = {
(12)    getWeatherData: function(zipcode){
(13)      GlobalGET('Weather').invoke('requestDataFromServer', zipcode);
(14)      return processWeatherData(GlobalGET('Weather').getProperty('temperature'));
(15)    },
(16)    requestDataFromServer: function(sendData){
(17)      var httpRequest = require('core/XMLHtttpRequest').module;
(20)      httpRequest.setProperty('onreadystatechange', function(){
(22)          GlobalGET('Weather').invoke('extractTemperature', httpRequest.getProperty('response'));
(24)      });
(25)      httpRequest.invoke('open', 'GET', serverUrl, true);
(26)      httpRequest.invoke('send', sendData); /*contact remote server*/
(27)    }
(28)  }
( )  exports.module = _module_; /*new statement added*/
```

Fig. 5. Code snippet from *Weather* module of the transformed DisplayWeather Jetpack extension. Only the statements from Figure 1 that are rewritten by Morpheus are shown.

Figures 4 and 5 show the rewritten statements and extracted user modules on applying Morpheus to our DisplayWeather extension (see Figure 1).

3.3 Policy Checker

Transformations on legacy extensions as applied by Morpheus greatly simplify enforcement of security policies on a per extension granularity. Morpheus supports both simple access control checks as well as complex stateful policy checks on sensitive browser resources and APIs managed by the core modules.

Security policies for preventing undesired accesses by the core modules are encoded in a separate Jetpack module named `PolicyChecker`, and all accessor methods in core modules must consult the `PolicyChecker` before actually granting access to the sensitive resources requested by a potentially compromised user module. To do so, Morpheus mandates that core modules place a trap in their accessor methods, as shown in Figure 3. `PolicyChecker` exports an API `check` to validate the request for accessing the sensitive resource by the user module. If the request does not conform to the extension's security policy, a violation is raised and the `PolicyChecker` simply blocks the requested access and returns an empty object.[2]

Since policies are encoded within the isolated `PolicyChecker` module and core modules can only invoke the `check` API to validate the access, Morpheus allows policies to be added or removed with no modification of the extension code.

[2] The supplementary materials contain an example of a security policy.

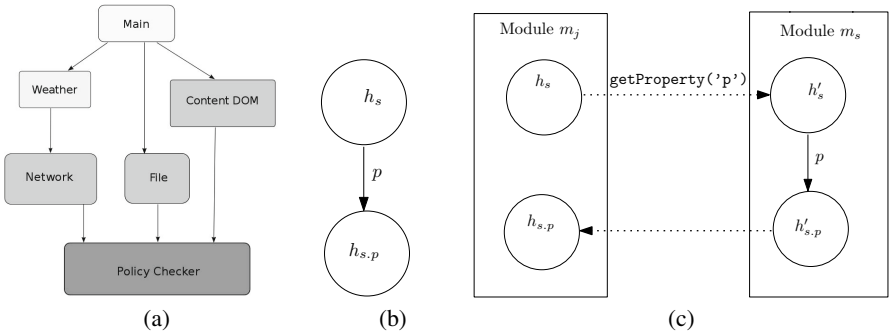

Fig. 6. (a) Module hierarchy in transformed DisplayWeather extension. Difference of heap map of property access of a sensitive object where h_ξ is the heap object for the expression ξ. (b) `s.p` in legacy extension (c) `s.getProperty('p')` in Jetpack. m_j is a user module, m_s is a module wrapping sensitive object `s`.

4 Security Analysis

A Jetpack extension's ability to limit the consequences of a breach depends on the structure of its modules and the security policies. Figures 6(b) and 6(c) show the effect of Morpheus's transformations in accessing property of sensitive object in terms of the heap model.

In a legacy extension when accessing a property `p` in sensitive object `s`, the heap object h_s for `s` and $h_{s.p}$ for `s.p` lies in the same address space, as shown in Figure 6(b). However when processed by Morpheus, `s.p` is rewritten as `s.getProperty('p')` and the heap object h_s for `s` does not have direct access to $h_{s.p}$, as shown in Figure 6(c). Instead, invoking the `getProperty` method gives it access to the actual heap object h'_s that has direct access to its property `p` heap object $h'_{s.p}$. The dotted line between $h'_{s.p}$ and $h_{s.p}$ denotes that (i) the latter is the wrapped version of the former object, and (ii) this relation is further protected by the policy enforcement mechanism. Note that both h'_s and $h'_{s.p}$ lie in a different module m_s, which is isolated from the module m_j corresponding to the transformed legacy code. Thus, if an attacker manages to compromise m_j he will not have direct access to the actual heap object from m_j.

Given the above heap model, we now analyze the security of a legacy extension transformed by Morpheus using several properties (enumerated in Table 3), provided in part by the Jetpack framework, Morpheus's transformation, and Morpheus's `PolicyChecker` for policy enforcement.

Let $P(m)$ denote the set of privileges that can be accessed by a module m. It is computed as follows:

$P(m) := (\bigcup_{m \to x} P(x)) \bigcup (\bigcup_{m \mapsto m^u} LP(m^u)) \bigcup (\bigcup_{m \mapsto m^c} P(m^c))$, where

$m \to x$ means module m has direct access to XPCOM interface x,

$m_i \mapsto m_j$ means module m_i imports module m_j,

U is the set of user modules m^u in an extension,

C is the set of core modules m^c in an extension and $U \cap C$ is $\emptyset$, and

$LP(m^u)$ denotes the set of privileges leaked from user module m^u

Table 3. Security properties

#	Provider	Property
P1	Jetpack	Each Jetpack extension is a hierarchical collection of modules that are isolated and share no state except that is explicitly exported using the `exports` construct.
P2		The set of privileges that can be manipulated and exported by a module depends on (i) user modules, and (ii) core modules it includes using the `require` construct.
P3		A module can import a privilege only when the Jetpack framework first loads the module. This implies that the module cannot dynamically extend its privileges at runtime.
P4		All Jetpack modules lie in chrome space and can contact with content Web page over an asynchronous message passing channel.
P5	Morpheus	Only core module can directly access XPCOM APIs. User modules can never directly access XPCOM APIs.
P6		Each core module encapsulates reference to only one XPCOM interface and does not have direct access to other XPCOM interfaces
P7		Core modules can not import any user module
P8		Each module exports only an opaque identifier and accessor methods, that can return either primitive values or instances of other modules
P9		Each module stores the reference to the sensitive object it encapsulates within another designated module, *i.e.,* all core modules share a common module to store sensitive objects.
P10	Policy Checker	Each core module can access a specific sensitive resource after being verified by security policy mediate the particular sensitive resource that a core module can access.

P3 together with **P2** guarantees that $P(m)$ can be statically determined and cannot be changed during execution, and thus prevents the attacker from creating and dynamically loading instances of other core modules inside the compromised core (or user) module m. **P5**, **P6** and **P7** limit the privileges $P(m)$ for any core module $m \in C$ to $(\bigcup_{m \to x} P(x)) \bigcup (\bigcup_{m \mapsto m^c} P(m^c))$. In case m is compromised, **P9** guarantees that the attacker only has access to the reference to the privileged object encapsulated by it (see Figure 6(b)), and no access to objects managed by other core modules, *e.g.,* m^c_j. This is because `core_module_table`, which stores the sensitive references for other core modules, does not support iteration and its accessor methods need an opaque identifier to return the sensitive reference. Since the opaque identifier itself is a reference, it is not possible for the attacker to manufacture the reference and access all sensitive objects.

For a user module $m \in U$, **P5** and **P8** guarantee that $\bigcup_{m \to x} P(x)$ is $\emptyset$ at all times. This implies that a user module cannot export references to privileged objects, because it has none. Therefore, we need not implement accessor methods for user modules, but Morpheus still keeps the same interface as it allows developers to conveniently enforce security policies on user modules. **P8** also guarantees that $\bigcup_{m \mapsto m^u} LP(m^u)$ is $\emptyset$ that makes $P(m)$ for any user module $m \in U$ equal to $\bigcup_{m \mapsto m^c} P(m^c)$. In other words, the privileges of a user module can be determined by inspecting privileges of the core modules it imports. Thus, the above properties ensure that for any module m, $P(m) \equiv \bigcup_{m \mapsto m^c} P(m^c)$ always holds.

The DisplayWeather extension with access to the user's file system and the network is an attractive target for Web attackers, who may want to steal sensitive user data, such as stored passwords, from the file system and send it over to an attacker controlled remote server. We now illustrate how Morpheus improves the security of the transformed DisplayWeather extension. Figure 6(a) shows the module hierarchy for the transformed Jetpack extension. Using the above formula and the transformed code (Figures 4 and 5), we claim that $P(m_{File}) \equiv \{file\}$, $P(m_{Network}) \equiv \{network\}$, $P(m_{Main}) \equiv \{file\}$, and $P(m_{Weather}) \equiv \{network\}$ holds even if these modules get compromised.

Unlike in the legacy DisplayWeather extension, **P4** guarantees that the modules in the corresponding Jetpack are isolated from the `content`. Assuming that the attacker has

(i) compromised the asynchronous message passing channel between the `content` and the `chrome`, and (ii) can infiltrate into the chrome space (that contains all the modules), we consider the case of a security breach in a user module $m_{Weather}$. The only privilege that the attacker gets is access to the network via the $m_{Network}$ module. Although we place no restriction on the nature of code that the attacker can evaluate within the extension, as listed earlier, **P3** restricts the powers of the attacker by disallowing him from loading a new core module $m_{LoginManager}$ (to read all stored passwords), as it was not requested by the compromised $m_{Weather}$ module at load time.

Due to the fixed module hierarchy in Jetpack extensions, the attacker cannot even trick m_{File} module (to read the password file) by only compromising $m_{Weather}$, and must also compromise m_{Main} or m_{File}. If we assume that the attacker has managed to infiltrate a core module m_{File}, then the only privilege he gets is $file$, *i.e.,* access to the file system. Similar scenario applies if the attacker has managed to infiltrate the core module $m_{Network}$. In each of the above cases, the attacker only gets access to the privileges available in the compromised module m computed by $P(m)$ and no more. This is in contrast to the legacy extensions where a breach in any portion of an extension enables the attacker to obtain access to any privileged object managed by the browser.

P10 further attenuates the authority of core modules. Let us assume that the attacker has compromised both the m_{Main} and $m_{Weather}$ modules, and also managed to modify the file path in `FileUtils.getFile` to the intended password file, and the URL for the remote server to one that is controlled by attacker. In such a scenario, the `PolicyChecker` will prevent the m_{File} and $m_{Network}$ core modules to read file other than `ProfD/zip.txt` from the file system and contact a remote server other than the legitimate weather server. Even if the attacker has compromised m_{File} and $m_{Network}$ module, the `PolicyChecker` will still prevent access to unauthorized resources.

We note that if the $m_{Weather}$ module was not extracted using Morpheus's transformations, $P(m_{Main})$ would have evaluated to $\{file, network\}$. In the absence of any security policy, compromising only m_{Main} module would have sufficed for the attacker. In other words, Morpheus does not worsen the security guarantees given by Jetpack framework. In fact, its module extraction based on the owning object algorithm along with the `PolicyChecker` make it harder for the attacker to mount a successful attack, by increasing the minimum number of modules that need to be compromised.

5 Implementation

We realized the entire Morpheus toolchain in about 13,400 lines of JavaScript (node.js [4]), of which about 10,500 lines were devoted to implement 100 core modules with wide ranging functionality. We used node.js to ease the implementation of the prototype. We leveraged Doctor JS [1], which also uses node.js as its backend, to implement our JavaScript code analyzer. Specifically, we added about 100 lines of code to customize Doctor JS for analysis of legacy extensions. Generation of Jetpack modules and rewriting of the global variables utilized the Narcissus [21] parser and decompiler to (i) rewrite the source ASTs, and (ii) convert the rewritten ASTs back to source code. This required about 4200 lines of JavaScript code. Finally, dynamic generation of the UI and subsequent packaging of the modules into a Jetpack addon required 900 and 100

lines of JavaScript code, respectively. Another 370 lines of shell scripts were required to automate the entire toolchain. Policy checker is implemented as a Jetpack module and requires only 150 lines of JavaScript code to encode all policies listed in Table 6.

The transformation of legacy extension into the corresponding Jetpack, and correct evaluation of chrome and content scripts in the transformed Jetpack posed several issues. We discuss a few of them here:

- *Content proxy.* A content proxy is required for mediating interaction between chrome and content scripts (see Section 2.3). The default content proxy implemented in the Jetpack framework was stateless, *i.e.,* execution of content scripts across different invocations of the proxy did not share any execution context. This stateless execution posed a problem since the transformed Jetpack requires multiple invocations to the proxy depending upon context switches, *i.e.,* from `chrome` to `content` and back (see line 36 in Figure 1). We overcame the problem by modifying the default content proxy to retain all execution state after initialization. The content proxy is initialized every time a new document is loaded.
- *Opaque identifiers.* Message exchange between the chrome and content scripts is asynchronous and is limited to transfer of primitive values and opaque identifiers only. Since object creation may also happen in the `content`, management of opaque identifiers must also be done in the content. We therefore inject the content proxy with scripts to manage opaque identifiers during its initialization.
- *Synchronous execution.* In order to retain the synchronous execution semantics as intended by the extension developer, Morpheus implements a synchronous execution protocol for evaluating object references in the `content`. Specifically, Morpheus utilizes the `processNextEvent` API defined on XPCOM's thread interface to implement the synchronous behavior by repeatedly processing the next pending event on the currently executing thread until it receives a response from the content process. This technique along with a stateful content proxy ensures that the transformed extension achieves synchronous execution semantics without blocking the CPU. However, this mechanism may affect the performance of the transformed extension if it makes numerous context switches between the `chrome` and the `content`.
- *Custom XPCOM interface.* Firefox allows extension developers to declare their own XPCOM components and register them with the extension architecture by packaging supporting JavaScript files, which implement the component interfaces, with the extension. Morpheus treats such JavaScript files as modules, redefines the components using helper methods provided by Jetpack and rewrites them like other JavaScript code in the legacy extension. All top level objects in extension scripts are also added to `global` module so that they can be accessed by the modules defining the XPCOM interface.

6 Evaluation

We evaluated Morpheus using four criteria: (i) correctness of the transformation, (ii) conformance to the principle of least authority (POLA), (iii) effectiveness of user module creation and (iv) effectiveness of policy-checker. We performed the evaluation using a suite of 52 legacy extensions (50 popular legacy extensions from Mozilla's addon

Table 4. Legacy extensions transformed using Morpheus and corresponding Jetpack statistics

Legacy Extension	Functionality	# Users
Amazon Search	Search in amazon.com using the right click context menu from any Website.	1,866
BlockSite	Blocks Websites and disables hyperlinks of user's choice.	214,173
Bookmark All	Bookmark all opening tabs quickly without any dialog.	5,304
Clear Cache	Clears the browser cache with one click	10,557
Clear Cache Button	Clears the browser cache.	44,843
Comment Blocker	Blocks or hides all unwanted comments on Websites.	1,415
Context Search	Expands the context menu's "Search for" item for all installed search engines.	67,070
Copy Link Text	Adds an option to the context menu to select the text of a link on right-click.	5,199
Copy Link URL	Copy the URLs of the selected links to clipboard.	13,025
Ebay Quick Search	Search in ebay.com using the right click context menu from any Website.	1000
Email This	Email link, title, and a selected summary of the Web page being viewed.	15,853
Empty Cache Button	Cache clearing made easy. One click.	53,048
Facebook Bookmark	Allow visiting Facebook Bookmarks by adding a special Button to Toolbar.	11,222
Facebook New Tab	Loads Facebook.com quickly when a new tab is opened.	7,439
Facebook Toolbar Button	Loads Facebook.com on clicking toolbar icon.	21,026
Facebook Touch Panel	Allow quick check Facebook Notifications and Messages by a touch Panel.	10,054
FlagFox	Displays a country flag depicting the location of the current Website's server.	1,296,480
FlashBlock	Blocks all Flash content from loading.	1,372,826
Go To Bing	Loads bing.com in a new tab when clicked on status-bar Bing icon.	139
Go To Google	Loads google.com in a new tab when clicked on status-bar Google icon.	15,700
Google Search By Image	Adds Google Search by Image context menu item for images.	45,838
Google Translator	Translates selected text or page into chosen language with a click or hot-key.	453,029
Google Viewer	Prompt to open supported documents with Google Docs Viewer.	1,472
Image Block	Adds a toggle button to conditionally block/allow images on Web pages.	22,147
ImageSearch	Adds a context-menu item for images to search Google for that image.	14,285
LEOs Dictionaries	Translates selected words/phrases with the help of LEOs Online Dictionaries	10,501
Leo Search	Searches selected words at dict.leo.org and opens the result in a new tab.	9,835
LibraryDetector	Detects which JavaScript libraries are being used on the current Web page.	1,590
Live IP address	Retrieves Live IP Address and displays in the status bar.	9,090
My Home Page	Load the homepage in a new Tab.	40,439
My Public IP Address	Show browser IP address.	2,959
New Tab Homepage	Load the homepage in a new tab; load the first in case of multiple homepages.	245,540
Open Bookmark (new tab)	Always opens new tab from bookmarks.	44,683
Open GMail (new tab)	Opens Google Mail Web page on a new tab.	22,107
Open GMail (pinned tab)	Opens Google Mail Web page on a new pinned tab in HTTPS mode.	10,092
Open Image (new tab)	Adds right-click context menu item for opening images in new tabs.	14,285
Place Cleaner	Replace the default "Print" button with Mozilla's "Print Preview" button.	21,878
Plain Text links	Open plain-text urls as links via context menu.	4,738
Print Preview	Replace the default "Print" button with Mozilla's "Print Preview" button.	37,966
Really Simple Sticky	Allow to add notes, reminders directly in the browser.	924
Right Click Link	Opens selected text in a new tab.	6,861
Search IMDB	Search the highlighted text at IMDB.	19,635
Show MyIP	Displays user's current IP address in the status bar.	11,239
Tab History Menu	Enables opening the history menu for a selected tab just by clicking on it.	7,237
TinEye Rev Img Srch	Adds a context menu to search for an image, where it came from, etc.	208,496
Twitter New Tab	Loads twitter.com quickly when a new tab is opened.	830
Twitter Toolbar Button	Loads twitter.com on clicking toolbar icon.	210
Web2Pdf Converter	Web page to PDF conversion tool.	42,185
YouTube Auto Replay	Enables automatic replay of a YouTube video or part of it.	26,478
YouTube IT	Search the selected Text in Youtube.	15,036
DisplayWeather	Displays weather of chosen location	N/A
Steal-login	Steal passwords and send to remote server	N/A

gallery (AMO) and 2 synthetic extensions) and then transformed them using Morpheus. Our dataset contained extensions that use common extension development technologies, such as JavaScript, HTML, XUL, CSS, etc., and did not contain any binary XPCOM component.

Correctness of transformation. We tested the correctness of the transformation by exercising the advertised functionality of each of the 52 extensions transformed with Morpheus. In each case, we enhanced the browser with the Jetpack extension being tested and observed the results of interaction with the extension's UI. Table 4 lists the extensions evaluated along with their functionality. The top 50 entries are for the real-world extensions whereas the bottom 2 correspond to the synthetic ones. For all cases the Jetpack extension was able to provide the advertised functionality of the original (legacy) extension.

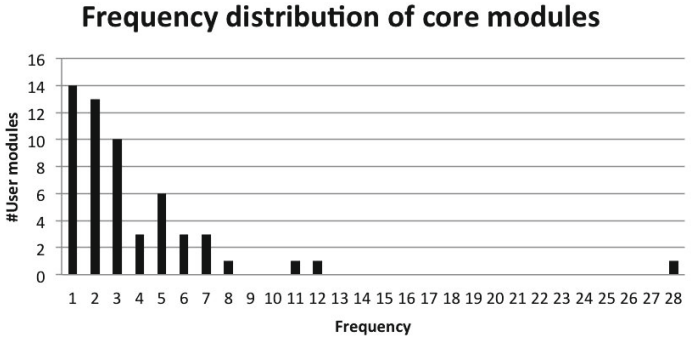

Fig. 7. Frequency of core modules in Jetpack user modules

FlagFox is one of the larger extensions that we transformed. It utilizes 28 core modules, and over 1307 lines of JavaScript (out of 3971 lines of extension code) are used to implement the UI. The remaining 2667 lines implement the core functionality of the legacy extension. We also observed that several extensions from our dataset had just a single user module after being transformed to Jetpack extension. Go To Google, Go To Bing, Steal-login are few instances of such case. This is due to the absence of any object definition or absence of property method invocations from objects defined in the legacy code. We also noticed the same Jetpack extension structure for TinEye Reverse Image Search entry even though the legacy code defines a top-level object. This is because it had all the functionality included in just that one object whose methods were invoked from event handlers.

Conformance to POLA. We used an off-the-shelf tool Beacon [17] to check whether modules in a Jetpack extension adhere to the principle of peast authority (POLA). Beacon detects whether a Jetpack module leaks references to privileged objects that it encapsulates. If so, any other code that `require`s this module will be able to directly access the privileged object without an explicit `require` of this object, thereby violating POLA. None of the 100 core modules leaks any object reference or violates POLA.

Privilege separation in user modules. We estimated the effectiveness of our user module extraction algorithm in approximating the ideal privilege separation by counting the number of core modules imported by each user module. The less the number of core modules accessed by a user module, the more effective is our module extraction algorithm in separating the privileges in extension code, as this corresponds to possible increase in the minimum number of modules that needs to be compromised to misuse multiple privileges.

We analyzed the user modules produced by Morpheus for all 52 Jetpack extensions and observed the frequency of the `require` invocation for various core modules within each user module. The goal is to demonstrate that user modules created using the owning object algorithm do not have access to large number of privileged objects as compared to legacy extensions. Figure 7 reflects the frequency distribution of core modules. We see that out of a total of 100 user modules across all the Jetpack extensions, there are

Table 5. List of Jetpack modules accessing multiple categories of core modules[3]. User modules created using owning object algorithm are named using random strings, except when they are either JavaScript code modules (JSMs) or the entry point of the extension *i.e.,* main module. Extensions not invoking any core module corresponding to XPCOM interfaces are omitted.

Jetpack	Module name	Categories					
		I	II	III	IV	V	VI
Amazon Search	M-1	✓					
BlockSite	M-1	✓			✓		
	M-2	✓					
	main	✓					✓
Bookmark All	M-1	✓					✓
	M-2					✓	
Clear Cache	main	✓					
Clear Cache Button	main	✓	✓				
CommentBlocker	appl (JSM)	✓					
	main	✓					
Context Search	M-1	✓					
	main	✓					
Copy Link Text	M-1	✓	✓				
Copy Link URL	M-1	✓	✓				
Email This	M-1	✓			✓		✓
Empty Cache Button	M-1	✓	✓				
Facebook Bookmarks	M-1	✓	✓	✓			
Facebook New Tab	M-1	✓	✓	✓			
	main	✓					
Facebook Toolbar Button	M-1	✓	✓	✓			
	M-2	✓	✓	✓			
Facebook Touch Panel	M-1	✓	✓	✓			
	M-2	✓	✓	✓			
FlagFox	flagfox (JSM)	✓	✓	✓	✓	✓	✓
	ipdb (JSM)				✓		
	main	✓					
FlashBlock	M-1	✓	✓		✓		✓
	M-2	✓					

Jetpack	Module name	Categories					
		I	II	III	IV	V	VI
Google Translator	M-1	✓					
Image Block	M-1	✓					
ImageSearch	M-1	✓		✓	✓		
LEOs Dictionaries	M-1	✓					
Leo Search	main	✓					
Live IP Address	main		✓		✓		
My Home Page	M-1	✓					
My Public IP	M-1		✓		✓		
New Tab Homepage	main	✓					
Open Bookmark (new tab)	main	✓		✓			
Open Gmail (pinned tab)	M-1			✓			
Open Image (new tab)	M-1	✓					
Plain Text Links	M-1	✓	✓				
Places Cleaner	M-1	✓	✓				✓
	M-2	✓	✓				
Really Simple Sticky	M-1	✓					
Search IMDB	M-1	✓	✓		✓		
Show MyIP	main				✓		
Tab History Menu	main	✓					
Twitter New Tab	M-1	✓	✓	✓			
Twitter Toolbar Button	M-1	✓	✓	✓			
YouTubeIT	M-1	✓					
TinEye Rev Img Srch	main	✓			✓		
Web2Pdf	M-1	✓		✓			
	main	✓		✓			
Dispaly Weather	M-1				✓		
	main				✓		
Steal Login	main				✓	✓	

56 modules with one or more accesses to distinct core modules. From the distribution, it is seen that around 14 modules use only one core module and as the number of core modules increases, the number of modules requesting multiple core modules decreases. We also note that there is one user module with 28 accesses to core modules. This user module is part of the FlagFox extension and is in fact a JavaScript code module (JSM) that was wrapped as a user module. Recall that JSMs are not partitioned into smaller modules because they are self contained code fragments (see Section 3.2).

Table 5 categorizes the usage of core modules corresponding to XPCOM interfaces across different categories, and we make four observations about it. First, most of the table is relatively sparse which indicates that user modules use related functionality. Second, almost all Jetpack extensions use core modules under the Application category and the reason is because they set user preferences. Third, since user modules created from JavaScript code modules, like flagfox in the FlagFox Jetpack, are just wrappers, they typically use core modules across multiple categories. Fourth, many Jetpack extensions which interact with content on Web pages, like DisplayWeather, do not explicitly invoke the core module `contentDOM` (see Section 3.2) responsible for access to the content objects. Instead they access properties of either chrome `window` or `gBrowser`,

[3] Core modules are grouped into 6 categories. Modules that access application or user preferences, create application threads, etc. are categorized under **I**. **II** contains core modules that represent browser neutral functionality such as access to timers and console. Modules facilitating access to content objects like `window` and `document` are grouped under **III**. Modules that handle browser permissions and cookies are grouped under **IV**, while those that access network, file system or storage come under **V**. The remaining modules are grouped under **VI**.

Table 6. List of policies checked for evaluation data set

Policy	Generic	# extensions
Contact only specified remote server	No	3
Access only files in profile directory as advertised	No	1
Cannot access preference branch other than its own	Yes	2
Cannot contact server if the extension has already accessed file system	Yes	1
Cannot contact server if the extension has already accessed LoginManager	Yes	1
Cannot contact server if the extension has access browsing history	Yes	1
Cannot contact server if the extension has access browser cache	Yes	2

which in turn invoke the `contentDOM` to make a transition to the `content`. Because of this implicit invocation, column entires in category **III** are empty for such Jetpack extensions.

Runtime policy checking. We evaluated the effectiveness of `PolicyChecker` at blocking attacks originating from misuse of the core modules. To do so, we encoded seven policies in the `PolicyChecker` module for the transformed extensions in our dataset. Table 6 lists these policies, which are classified as being either generic or extension-specific. The first three policies enforce fine-grained access control over extension resources, and the remaining policies are stateful. Of the extensions in our dataset, only Steal-login exhibits malicious activity, while the others are benign and do not violate the policies in Table 6. Thus, to verify that `PolicyChecker` can actually identify and block violations in core module, we introduced synthetic violations in benign extensions. We did so by appending additional code within the user modules of the benign but transformed extensions to trigger policy violations. The third column in the table lists the number of extensions that were used to check such synthetic violations of the corresponding policy. In each case, we observed that `PolicyChecker` was able to identify the violation and block the undesired operation in the core module. In our experiments, we refrained from checking any policy for an extension if it can potentially block the advertised functionality. For example, we did not apply policy to block network access after file system access for the DisplayWeather extension, as the extension contacts a weather server after reading 'zip.txt' from the file system, which is its advertised functionality. We do envision developer assistance when encoding such policies.

We now list specific observations on applying Morpheus over legacy extensions.

(1) An extension from our dataset CommentBlocker[4] installs event handlers that manipulate objects from both `chrome` and `content` to achieve its advertised functionality. Specifically, it installs two mutation event listeners (for `DOMNodeInserted` and `DOMNodeRemoved` events) in the `content` while their handlers are declared in the `chrome`. Execution of such event handlers invokes frequent invocations to the synchronous execution mechanism due to context switches between the `chrome` and `content`. Since the Jetpack framework disallows direct access of references across the `chrome`/`content` boundary, Morpheus transforms the handler defined in the `chrome` to operate using opaque identifiers for the `event` object (which is passed implicitly to all handler functions). Creating opaque identifiers for `event` attributes like `target` and `originalTarget` allows most functionality, but prevents operations such as `evt.target instanceof HTMLDocument`. This is because the Jetpack framework

[4] CommentBlocker:https://addons.mozilla.org/en-US/firefox/addon/commentblocker/

itself does not provide support for all objects available in the legacy Firefox extension architecture. For example, comparison of object instances against `HTMLDocument` and other `HTML` elements using the `instanceof` operator does not succeed in the Jetpack framework. Thus, legacy extension using such comparisons must be rewritten to use alternate comparisons (such as `Ci.nsIHTMLDocument` and `Ci.nsIHTMLElement`).

(2) The interface definitions for most XPCOM APIs inherit from other interfaces. For example, the `nsILocalFile` interface inherits from `nsIFile`. `QueryInterface` [22] is a construct that allows JavaScript to perform runtime type discovery and identify the interfaces supported by an object. Thus, on instantiating an object of type `nsILocalFile`, the object can perform a `QueryInterface` to access methods and properties defined on the `nsIFile` interface as well. With the core modules exporting only accessor methods, `QueryInterface` on module objects would be incorrect. To correctly implement the behavior of `QueryInterface`, the getter method in `core_module_table` maintains a linked list of objects which were `QueryInterface'd` on a module object and on every property access, it traverses the list and returns the object on which the property was defined.

(3) If an XPCOM API returns an instance of a string object, its core module returns a wrapped string object that exports an opaque identifier and the three accessor methods (*i.e.,* `getProperty`, `setProperty` and `invoke`). Since this wrapped string object cannot be directly used for string operations like concatenation, Morpheus appends an additional `toString` property on the wrapped string object.

In its current form, Morpheus is constrained mainly due to Narcissus and Doctor JS. The Morpheus toolchain uses both these tools during different phases of its operation. Both Narcissus and Doctor JS are under active development and do not support all JavaScript constructs and features. For example, Narcissus does not support various forms of the `let` block, array comprehension, destructuring, generators, etc. Doctor JS uses the CFA2 algorithm [32] for JavaScript implemented atop Narcissus. Doctor JS also does not support a number of JavaScript statements. For example, it throws exceptions when performing string concatenation via the += shorthand operator, or if the loop variable is not defined explicitly within the `for` loop itself. We are actively working to remove such limitations by porting Morpheus to a more stable platform, like SpiderMonkey [23], and allow evaluation of more complex extensions.

7 Related Work

There has been much interest recently in the research community to improve defenses against vulnerable and malicious browser extensions. This paper presents an automated approach to port legacy extensions to secure, modern platforms and to our knowledge, Morpheus is the first tool to do so.

Securing browser extensions. The Jetpack framework is similar to the Google Chrome extension architecture [9] which encourages a modular design. Recent work [11, 19] explores the latter to highlight its deficiencies in developing secure Chrome extensions.

VEX [8] implements a flow- and context-sensitive static analysis of JavaScript to study vulnerabilities in legacy Firefox extensions. Beacon [17] performs information-flow for modular JavaScript extensions and is designed to detect poor software

engineering practices in modules, *i.e.,* violation of POLA or leaked capabilities across module interface. Sabre [13] and Djeric and Goel [14] both present dynamic information flow tracking system to detect extensions that can leak sensitive browser data. IBEX [15] is a framework for specifying fine-grained access control policies guarding the behavior of monolithic browser extensions, but requires extensions to first be written in a dependently-typed language (to make them amenable to verification), following which they are translated to JavaScript.

Runtime policy enforcement has also been applied to prevent extensions from leaking sensitive data and limiting extension privilege in [27, 31]. Even though the approach presented in [27] is more light-weight than [31], both techniques require modifications to the browser. Similar to Morpheus, [27] wraps all accesses to XPCOM interfaces in legacy extensions to validate the operations with regard to security policies specified on the extension. In contrast, our main goal in wrapping privileged objects in individual modules is to adhere to Jetpack's security principles and limit the damage to only the compromised module. The extension architecture also enables embedding fine-grained security policy enforcement without modifying browser or Jetpack runtime. Morpheus improves security of extensions by both porting to Jetpack and enforcing policies.

Privilege separation. Morpheus is most closely related to Privtrans [10] and Swift [12]. Privtrans automatically integrates privilege separation into legacy source code using context switching between a secure monitor and an untrusted slave. Swift defines a principled approach to build secure web applications by partitioning the source code. Morpheus uses both approaches. It defines an evaluation context for object references, as either `chrome` or `content`, and switches contexts when execution of a JavaScript statement contains references from both contexts. This context switching approach is needed because the Jetpack framework is restrictive and does not allow placement of content code in `chrome` or vice-versa. Morpheus differs from both Privtrans and Swift and several other privilege separation mechanisms [28, 18, 26, 33, 34], because it is entirely automatic and does not require any user annotations to accomplish partitioning. A new architecture is proposed in [7] to achieve privilege separation for HTML5 web applications including browser extensions. Morpheus is orthogonal to [7] and ports legacy code to the Jetpack framework that mandates chrome-content privilege separation.

8 Conclusion

We present Morpheus, a streamlined mechanism to port legacy Firefox extensions to the more secure Jetpack framework. It utilizes module isolation provided in Jetpack framework to overcome challenges in code partitioning and secure module construction. Transformation applied by Morpheus enables fine-grained policy enforcement on ported Jetpack extension. We evaluate Morpheus with a suite of 52 legacy extensions and show that the automatically transformed extensions are secure by construction.

Acknowledgments. This work was funded in part by AFOSR grant FA9550-12-1-0166 via subaward 4628-RU-AFOSR-0166. We thank Santosh Nagarakatte, Chung-chieh Shan, and the anonymous reviewers for comments on early drafts of this paper.

References

1. Doctor, J.S.: http://doctorjs.org/
2. Jetpack, https://wiki.mozilla.org/Jetpack
3. JSON, http://www.json.org/
4. node.js, http://nodejs.org/
5. Opera extensions, http://dev.opera.com/extension-docs/
6. Safari extensions, https://developer.apple.com/library/safari/documentation/Tools/Conceptual/SafariExtensionGuide/Introduction/Introduction.html
7. Akahawe, D., Saxena, P., Song, D.: Privilege separation in HTML5 applications. In: USENIX Security Symp. (2012)
8. Bandhakavi, S., King, S.T., Madhusudan, P., Winslett, M.: Vetting browser extensions for security vulnerabilities with VEX. CACM 54(9) (September 2011)
9. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: Network and Distributed Systems Security Symp. (2010)
10. Brumley, D., Song, D.: Privtrans: automatically partitioning programs for privilege separation. In: 13th USENIX Security Symp. (2004)
11. Carlini, N., Felt, A.P., Wagner, D.: An evaluation of the google chrome extension security architecture. In: USENIX Security Symp. (2012)
12. Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., Zheng, X.: Secure web applications via automatic partitioning. SIGOPS Oper. Syst. Rev. 41(6) (2007)
13. Dhawan, M., Ganapathy, V.: Analyzing information flow in javascript-based browser extensions. In: Annual Computer Security Applications Conference (2009)
14. Djeric, V., Goel, A.: Securing script-based extensibility in web browsers. In: USENIX Security Symp. (2010)
15. Guha, A., Fredrikson, M., Livshits, B., Swamy, N.: Verified security for browser extensions. In: Proc. of IEEE Symp. on Security and Privacy (May 2011)
16. Hardy, N.: The confused deputy (or why capabilities might have been invented). SIGOPS Oper. Syst. Rev. 22(4) (October 1988)
17. Karim, R., Dhawan, M., Ganapathy, V., Shan, C.-c.: An analysis of the Mozilla Jetpack extension framework. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 333–355. Springer, Heidelberg (2012)
18. Kilpatrick, D.: Privman: A Library for Partitioning Applications. In: USENIX Annual Technical Conference, FREENIX Track (2003)
19. Liu, L., Zhang, X., Yan, G., Chen, S.: Chrome Extensions: Threat Analysis and Countermeasures. In: Network and Distributed Systems Security Symp. (2012)
20. Mozilla. Add-on SDK, https://addons.mozilla.org/en-US/developers/docs/sdk/latest/
21. Mozilla. Narcissus, http://mxr.mozilla.org/mozilla/source/js/narcissus/
22. Mozilla. Query Interface, https://developer.mozilla.org/en-US/docs/XPCOM_Interface_Reference/nsISupports#QueryInterface
23. Mozilla. Spidermonkey, https://developer.mozilla.org/en/SpiderMonkey
24. Mozilla Developer Network. Electrolysis, https://wiki.mozilla.org/Electrolysis
25. Mozilla Developer Network. XPCOM, http://developer.mozilla.org/en/XPCOM
26. Myers, A.C.: Jflow: practical mostly-static information flow control. In: ACM Principles of Programming Languages (1999)

27. Onarlioglu, K., Battal, M., Robertson, W., Kirda, E.: Securing legacy firefox extensions with SENTINEL. In: Rieck, K., Stewin, P., Seifert, J.-P. (eds.) DIMVA 2013. LNCS, vol. 7967, pp. 122–138. Springer, Heidelberg (2013)
28. Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: 12th USENIX Security Symp. (2003)
29. Addon SDK. Content proxy, `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/dev-guide/guides/content-scripts/accessing-the-dom.html`
30. Simon Willison. Understanding the Greasemonkey vulnerability, `http://simonwillison.net/2005/Jul/20/vulnerability/`
31. Ter Louw, M., Lim, J.S., Venkatakrishnan, V.N.: Enhancing web browser security against malware extensions. J. Computer Virology 4 (2008)
32. Vardoulakis, D., Shivers, O.: CFA2: a context-free approach to control-flow analysis. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 570–589. Springer, Heidelberg (2010)
33. Zdancewic, S., Zheng, L., Nystrom, N., Myers, A.C.: Secure program partitioning. ACM Trans. Comput. Syst. 20(3) (August 2002)
34. Zheng, L., Chong, S., Myers, A.C., Zdancewic, S.: Using Replication and Partitioning to Build Secure Distributed Systems. In: IEEE Symp. Security & Privacy (2003)