TECHNIQUES AND TOOLS FOR SECURE WEB BROWSER EXTENSION DEVELOPMENT

BY

REZWANA KARIM

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Vinod Ganapathy

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2015

ABSTRACT OF THE DISSERTATION

TECHNIQUES AND TOOLS FOR SECURE WEB BROWSER EXTENSION DEVELOPMENT

by REZWANA KARIM

Dissertation Director: Vinod Ganapathy

Many modern application platforms support an extensible architecture that allows the application core to be extended with functionality developed by third-parties. This bootstraps a developer community that works together to enhance and customize the basic functionality of those platforms. To ease development of such extensions, these platforms expose an API that third-parties can use to implement their functionality. For instance, Web applications make use of the browser's Document Object Model (DOM) API, smart phone applications use the mobile platform's SDK and browser extensions use the extension API. These APIs usually endow extension developers with privileges to access various system resources. However, to isolate the platform from any new security threats caused by these untrusted extensions, the API must ideally restrict extensions' authority. Thus, an important challenge is to simplify extension programming for the third-party developers while ensuring that these extensions do not compromise the security of the application core.

This dissertation seeks to address the above issues in the context of Web browser extensions. It presents algorithms and tools to facilitate secure Web browser extension development. In particular, it makes the following two contributions.

First, it studies and characterizes the security of a modern Web browser extension architecture, the Mozilla Jetpack framework — proposes solutions to improve the security of the architecture and extensions developed on top of it. It presents Beacon, which leverages JavaScript-level information flow technique to detect unsafe programming practices in browser extensions. Upon analyzing 68000 lines of JavaScript code from modern extension framework and real world extensions, Beacon found 36 instances of potentially unsafe programming practices.

Second, it addresses the problem of porting unsafe legacy extensions to modern, privilege-separated extension architectures. It presents Morpheus, which applies program analysis and software engineering techniques that refactor legacy vulnerable extensions for use with modern extension frameworks, the Jetpack framework in particular. Morpheus also enables fine-grained control over extensions via a runtime policy enforcement engine. Morpheus has been applied to successfully port 52 legacy Mozilla extensions to the Jetpack framework.

Acknowledgements

First and foremost, I would like to thank my graduate advisor, Professor Vinod Ganapathy, for his constant support, guidance and motivation. His insights and feedbacks have been instrumental in shaping several ideas of my thesis. Without his patience and persistence this thesis would not have been a reality. At a critical juncture of my personal life, he has been compassionate and flexible, which has allowed me to attempt a work-life balance during the final year of my PhD. I am also thankful to Professor Liviu Iftode, Professor Ulrich Kremer, Professor Santosh Nagarakatte, and Professor Long Lu (Stony Brook University) for their invaluable suggestions on improving my dissertation. I have benefited greatly from my interaction with them.

I have had the honor of working closely with Dr. Nishant Sinha (IBM Research) and Professor Chung-chieh Shan (Indiana University). Collaboration with them has greatly influenced my thought process as a researcher. Their passion and dedication towards work has inspired me to work harder. I have also had the privilege to spend a Summer at Mozilla and work with Dr. David Herman and the Mozilla Jetpack team whose insights and discussions were crucial to my research on browser extensions. I would also like to thank Dr. Randy Smith (Sandia National Labs) for his contribution to my research on network security.

I feel fortunate to have shared my time at Rutgers with several excellent colleagues. I would like to express my gratitude to my peers, Mohan Dhawan and Liu Yang, for contributing their expertise in my research. Working with them in multiple research problems has been a rewarding experience. I am grateful to many other members at Disco-Lab, Rutgers University, especially Shakeel Butt, Amruta Gokhale, Nader Boushehrinejadmoradi, Lu Han, Daeyoung Kim, Hai Nguyen, Ruilin Liu, and Daehan Kwak for their valuable discussions and feedback on my research projects and presentations.

Coming to the USA on my own and living far away from my family, for the first time, was

not easy. I am indebted to my friends at Rutgers for their emotional and logistical supports over the years. They have hardly let me feel that I am in a foreign country and have helped me to sail through this long difficult journey. Especially Amruta Gokhale, Anwesha Chaudhury, Shreyasee Mukherjee, Priya Govindan, Chathra Hendahewa, and Rummana Rahman – I feel blessed to have them through thick and thin in my graduate life. Friends from my undergraduate university and school friends, living in various parts of the world, have been distant sources of support and inspiration. I would also like to thank the Bangladeshi families who have extended their helping hand when needed. Special thanks to my friend Sayema and her parents for making me feel at home whenever I visited them in New York.

My deep gratitude to all the people from various stages of academic life who have taught me academic materials and programming. I am also thankful to those who have helped me in developing driving and culinary skills. Surviving alone in the USA would have been extremely difficult otherwise.

My biggest source of strength and inspiration is my family. My parents, Md. Rezaul Karim and Ruby Akter Begum, have always taken a keen interest on my academic progress, and put my academic need ahead of their own comfort and happiness. Without their unconditional support, I would not have been able to reach this level. Words are not enough to express my gratitude to my mother. She has always allowed me to follow my dreams and truly been a protective shield. She has never stopped believing in me and reinforced my confidence, even when I would occasionally be plagued with self-doubt. I owe all my achievements to her. My brother, Md. Tahsin Karim Nabil, though much younger than me, has been extremely helpful in taking care of my parents and giving them much needed support while I immersed myself into work. My grandparents, uncles, aunts, cousins and even distant relatives have always taken pride in my academic achievements and their affection has been truly invigorating. Their excitement on my PhD accomplishments is particularly heartwarming. I thank my youngest uncle for instilling the practice of being self-challenging since childhood. I am also thankful to my husband, Istiaque Ahmed, for his patience and support while I was working on my dissertation.

Finally, I would like to thank the Almighty for the successful completion of my PhD.

Dedication

To my mother, Ruby Akter Begum, for everything.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation considers the problem of building secure Web browser extensions. In particular, it makes two contributions. First, it studies and characterizes security problems in Jetpack, the modern Web browser extension framework by Mozilla, and proposes a solution to improve the security of modern extensions by automatically detecting unsafe programming practices. Second, it develops methods to assist extension developers build code that adheres to security principles. Violations of these principles could lead to exploitable vulnerabilities. To further strengthen the security of modern extensions built atop Jetpack, it proposes runtime policy enforcement that allows fine-grained access control and blocks potentially dangerous information flows. The methods developed in this dissertation have been prototyped in two tools, Beacon and Morpheus. Beacon detects unsafe programming practices in real world JavaScript-based browser extensions and the extension framework itself. Morpheus is designed to automatically transform legacy extensions by porting them to the Jetpack framework that provide stronger security guarantees.

1.1 Extensibility of the Web Browser

The Web browser has evolved into a complex computing platform, and in certain cases has become the ”de-facto” operating system for Web applications. Part of the credit can be attributed to its extensible architecture that allowed third party developers to continuously enhance and customize the basic functionality of the browser via applications that are known as browser extensions. Some of the popular extensions eventually became integral parts of the browser itself. Not surprisingly, most modern browsers support such third-party extensions and nurture an extension developer community to escalate development of extensions that enrich the browser ecosystem. Browser extensions offer a wide range of functionality, for example, customizing

Web page look and feel, debugging Web applications, intercepting and analyzing network traffic, manipulating security and privacy sensitive data and interacting with a remote Web service like Google Translate. Such extensibility popularizes the browser to end-users, thereby attracting more developers to embrace the platform. 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 extensions. This dissertation focuses on scriptable extensions for the Mozilla Firefox browser. Such scriptable extensions, written mostly 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 July 2015, around 14000 extensions, supporting a wide variety of functionalities, are available for Firefox via the Mozilla extensions gallery [14] with popular extensions often used by millions of users.

1.2 Browser Extension Security

To support rich functionality in extensions, most modern Web browsers export privileged APIs allowing extension developers to access many sensitive resources, for example file system, password, cookies, network and browser cache. These JavaScript based extensions are fundamentally different from Web applications that execute JavaScript code in front-end. Code that executes within a Web page is often tightly sandboxed by the browser, *e.g.,* using the same-origin policy [18], and does not have access to privileged browser APIs whereas JavaScript code in extensions run with the privilege of hosting principals *i.e.,* a Web browser. Unfortunately, browser extensions do not undergo the same quality control as the rest of the browser, and are riddled with vulnerabilities. Hence, such capability *i.e.,* unfettered privileged access to resources, can be misused by attacks directed against vulnerable ones. As a consequence, compared to Web applications, these extensions pose more threat to end-users — the benign-but-buggy extensions can be exploited by remote attackers to take control over the entire Web browser, and steal sensitive end-user data.

Foreseeing threats to browser security, vendors have maintained a site requiring authorization, somewhat equivalent to a marketplace for mobile apps, where the developers can make their extension available for users to download. Users are strongly recommended to install the

extensions that are only available in the marketplace as the browser vendors can monitor them for performance and security as they are uploaded to the gallery. Browser vendors typically perform a security review of extensions to check adherence to vendor-prescribed security best practices before being featured in the marketplace. If an extension fails the review process, it is returned to the developer to fix the reported issues, and hence the entire process involves a development-review-development cycle before being finally released for public use. However, this review process is often conducted manually and adds additional delays in publishing extensions which can be frustrating for developers. Moreover, the process largely depends on developers' expertise and meticulous effort. Extension security thus becomes a double-edged sword for browser vendors — it is strictly required to safeguard the browser platform from extension vulnerabilities, on the contrary, it raises the barrier of adoption of the extension architecture among the third-party developers. Thus, it is desirable to develop faster review process, automated to the extent possible. Providing the extension developer with tools to detect vulnerabilities upfront can further speed up the process.

1.3 Two Models of Extension Development

Since its inception, scriptable extensions for the Firefox browser have been primarily developed using open technologies such as HTML, CSS, JavaScript and XUL [20]. In general, extensions can be developed using two different models : (1) legacy and (2) modern.

Extensions following the legacy model make heavy use of XUL and invoke low-level XPCOM [66] APIs to access sensitive system resources. These legacy extensions have a monolithic architecture, where extension code and code interacting with Web page content execute in a unified JavaScript heap. The legacy API lacks high-level abstractions and extension authors are required to write their code from scratch, directly accessing the XPCOM interface to perform privileged actions. Such an approach, although lacking modularity, endows developers with the flexibility of implementing diverse functionalities. On the flip side, it provides too much authority to each extension. The unified JavaScript heap raises the risk of shared references that could easily lead to privilege escalation. An exploitable vulnerability anywhere in the extension typically exposes the entire XPCOM interface to the attacker.

Several solutions, both using offline analysis [23, 24] and runtime detection [35, 37], have previously been proposed to detect vulnerabilities in legacy extensions. These solutions mainly attempts to diagnose and, if possible, restrict certain sets of information flows which potentially leak sensitive data. The offline analysis based solutions, which predominantly employ a static analysis of the extension JavaScript code, emphasize detecting unsafe code patterns that are likely to leak sensitive data. Besides being unsound and imprecise, static analysis alone is not sufficient to track flow of data at a fine-grained level. Runtime detection based solutions, in contrast, aim to restrict potentially dangerous information flow based on security policies and employ a runtime policy enforcement. However, in the absence of any defense-in-depth mechanism and properly structured modular code, achieving an adequate level of security requires extensive policy enforcement in legacy extensions. This approach effectively increases the runtime overhead and deteriorates the perceived performance at the user end (not to mention being difficult to implement correctly). Moreover, enforcing fine-grained security policies in legacy extensions is non-trivial and the intricacies include modifications of the actual extension code, the extension API and often the browser itself [67, 74]. The underlying reason is the monolithic structure of legacy extensions that leads to an undesirable mix of program logic with security logic. With a unified heap and lack of any isolation primitives in the JavaScript language, extension developers must consciously and carefully restrict access to critical functionality. Weaving policies requires a thorough understanding of the extension code and implications of security policies, and expertise to reason about placement of checks. The non-modular structure is particularly unfavorable for deploying a modular, extensible policy checker framework, where extension code can be oblivious of addition or removal of security policies. Finally, browser level support would be required to accommodate any policy that necessitates distinction of objects coming from different trust origins.

Securing legacy extensions is thus a challenging task for all involved parties – developers, security tools designers and browser vendors. The fundamental problem lies in the extension architecture itself — extensions implemented using heterogeneous technologies and containing code from different trust levels execute in the same program heap. The monolithic architecture not only leads to vulnerabilities in legacy extensions due to unsafe programming practices, but the mishmash of technology is also intimidating to novice developers.

Web browser vendors have therefore recently developed new extension frameworks exporting better abstractions, and aimed at better isolating extensions while still allowing them access to privileged browser state. Extensions that run on top of these modern frameworks follow a different programming model compared to legacy extensions. Examples of such frameworks include the Google Chrome extension architecture and Mozilla's Jetpack extension framework for Firefox.

The Jetpack framework focuses on easing the extension development process with an emphasis on modular development and security. In Jetpack, each extension is a hierarchical collection of modules where each module explicitly requests the capabilities that it requires, *e.g.,* access to specific sensitive resources. Jetpack therefore aims to confine the effect of a vulnerability to an individual module by (1) structuring extensions as a collection of isolated modules that communicate via clearly defined interfaces and (2) attempting to enforce security principles like privilege separation and the principle of least authority (POLA) [70], violation of which could trigger privilege escalation. In other words, Jetpack adopts the strategy of defense-in-depth — a thorough arsenal of isolation between extension code and untrusted Web content, a hierarchical collection of isolated modules, and adherence to security principles. The responsibility to secure extensions is distributed in this layered defense mechanism. Jetpack takes an architectural approach to guarantee the first layer of defense — process-level isolation between JavaScript running in the Web page context and in the extension context. The next two layers — hierarchical organization of isolated modules and adherence to security principles, form the second line of defense and are dictated by the structure of the extension code. Additionally, satisfying structural requirements eases reviewing modern Jetpack extensions through static reasoning about security properties at the module level.

Despite the introduction of modern extension frameworks, legacy extensions are still supported by Firefox due to their large user-base, with the popular ones being actively used by millions of users on a daily basis. Moreover, some developers continue to favor the legacy model due to their long familiarity with the programming model and the flexibility of performing privileged actions without being restricted by constraints imposed in modern extension frameworks.

1.4 Motivation

Adherence to the security principles in Jetpack extensions, which follows a modern model of extension development, can give a stronger security guarantee compared to legacy extensions. However, structuring extensions to conform to these security principles requires developers' expertise and a deep understanding of the security implications triggered by violations of security principles. As shown by our study of more than 600 Jetpack modules, obtained from the Jetpack codebase and 359 real-world Jetpack extensions, such violations are quite common. There are several instances where the recommended programming principles are not followed by developers resulting in capability leaks through the module interface. Such leaks, often inadvertent and hard to detect via manual inspection, induce overprivileged modules that violate security principles. This nullifies the benefit that Jetpack attempts to provide in limiting the effect of vulnerabilities via module isolation and static reasoning about capabilities at the module level. Given that most developers are not security experts and even the highly skilled developers can make mistakes that can thwart security of the browser, modern extensions too have to undergo a security review at the reviewer's end. As highlighted earlier, it would be more pragmatic to automatically detect security violations. However, existing solutions to statically detect unsafe programming practices in extensions are only applicable to legacy extensions, and that too to uncover leaks of sensitive data, and cannot be directly leveraged to detect module level capability leaks in modern extensions. Casting the solutions for legacy extensions to make them amenable for analyzing modern extensions would require a substantial amount of effort. It would necessitate modelling a different set of APIs, tracking capability instead of data and finally addressing the core structural differences between legacy and modern extensions as discussed later in this chapter. Further, additional analysis would be required to identify instances of modules that ask for privileges but never utilize them in the code.

While adherence to recommended best practices can contain the effects of vulnerabilities in a modern extension, these best practices do not safeguard against attacks such as confused deputy [47] attacks. To mitigate such attacks, extensions must be protected with another layer of defense in the form of fine-grained control over the actions of extensions. Security policies can further attenuate the authority of extension code in case the attacker tries to misuse the

privileges given to the compromised extension. The modular structure of modern extensions facilitates embedding the policy checker without affecting the actual extension code. Note that runtime policy enforcement can possibly degrade the end-user experience by introducing runtime performance overhead. However, we show that overhead can be minimized by strategically crafting the policy and inserting checks only at limited number of program points. Together with runtime policy enforcement, conformance to security principles can improve the security guarantee of modern extensions compared to legacy extensions.

Unfortunately, the current practice is to develop such modular extensions from the ground up, adhering to the programming disciplines that the frameworks enforce. Legacy extensions cannot directly execute atop modern extension frameworks and gain the benefit they provide due to two key factors: (1) structural and architecture-level incompatibility and (2) differences in support for various technologies (extension UI and binary components). Therefore, in order to obtain the same security guarantees as in a modern extension, a legacy extension must be ported to the new framework. However, doing so manually would be expensive and time-consuming. The situation is exacerbated by the prevalence of thousands of popular, but unsafe, legacy extensions and the persistent use of the legacy model by a section of extension developers. This huge corpus of legacy extensions must be ported to the modern Jetpack framework, not only for security purposes, but also for design compatibility with future multi-process browser architectures. Legacy extensions need to be transformed in a way that they conform to the recommended security principles while preserving program semantics. The underlying challenge here can be attributed to the key differences between the programming models of legacy and the Jetpack extension framework:

1. *Structure of extension code:* Legacy extensions offer a unified JavaScript program heap for all extensions whereas Jetpack offers the abstraction of isolated heaps for individual modules.

2. *Nature of communication between different trust levels:* In legacy extensions, both privileged extension code and unprivileged Web page content lies in same address space and therefore can communicate with each other in a synchronous manner. In contrast, they execute in two different processes in Jetpack extensions and interact via an asynchronous

message passing protocol.

This dissertation addresses these challenges involved in developing secure extensions and supports the following problem statement.

> **Problem Statement**: To safeguard the browser from extension vulnerabilities, modern extension frameworks highly depend on third-party extension developers' expertise and meticulous effort in adhering to certain programming and security principles.

1.5 Simplifying Secure Extension Development

This dissertation takes the above problem into consideration and proposes techniques and tools for developing secure extensions. The target user of these tools are extension developers and reviewers, not the end-user.

We study Jetpack, Mozilla's modern extension framework, to characterize the problems that lead to violations of POLA and privilege separation. We model this problem as a capability leak detection problem. A leaked capability endows another module with privilege that it did not explicitly ask for which clearly violates the recommended security principles. We present Beacon, a capability flow analysis tool, to automatically detect such leaks and any violation of these principles during extension development. The key idea is to statically analyze the extension module code to track the flow of capability from its point of origination, *i.e.,* sensitive resource access to the module interface. Our extensive evaluation of the Jetpack framework and the real-world Jetpack extensions detected 36 capability leaks. Often, these leaks are not required for extension functionality and hard to detect via manual inspection. This observation suggests that even heavily-tested production-quality code may contain capability leaks. Beacon can help extension developers to uncover these leaks early in the development phase and ensure that their code conforms to the recommended security principles. Similarly, reviewers can utilize it in the extension vetting process before making them available to the public.

We address the significance of retargetting legacy extensions to modern frameworks and

identify the challenges involved in it. These challenges, which mainly revolve around refactoring legacy extension code and constructing secure modules while retaining the UI and functionality, stem from the differences in programming models and the necessity to conform to certain security principles. We present Morpheus, a static analysis and transformation tool that allows legacy Firefox extensions to be systematically ported to Mozilla's Jetpack framework. Morpheus statically analyze a legacy extension code and partition it into Jetpack modules that satisfy POLA and privilege separation. Each module encapsulates objects corresponding to sensitive XPCOM APIs and enables accessor methods which provide the required API functionality. Morpheus empowers developers to automatically port their legacy extensions to Jetpack and preserve their earlier investment in developing them. It also allows extension developers to write legacy-style imperative code and systematically convert it to a Jetpack extension in order to benefit from the enhanced security guarantee offered by the framework. We evaluate Morpheus with a suite of 52 legacy extensions and show that the way Morpheus transforms the legacy extensions greatly improves their security guarantee.

Transformations on legacy extensions, as applied by Morpheus, also allows enforcement of fine grained security policies without any modification to the browser runtime or additional instrumentation of the extension code. We propose a policy checker framework for Jetpack which allows runtime policy enforcement at a fine-level of granularity to control sensitive resources accesses. The modular and extensible architecture of a Jetpack extension, together with the policy checker, allows the developers to seamlessly add or remove security policies without affecting the rest of the code. Policies are designed and checks are inserted in such a way that the policy enforcement does not incur any perceivable runtime overhead.

1.6 Summary of Contributions

The thesis this dissertation supports is:

> **Thesis Statement**: Automated program analysis techniques can identify violations of security principles in extensions and extension framework code. Automated program transformations can be used to port legacy browser extensions to modern extension frameworks.

In light of the above thesis statement this dissertation makes the following contributions:

- We study Jetpack, Mozilla's modern extension framework, to characterize the problems that lead to violations of POLA and privilege separation (Chapter 3). We model this problem as a capability leak detection problem. We present Beacon, a capability flow analysis tool to automatically detect such leaks and any violation of these principles during extension development.

- We address the challenges involved in porting a huge corpus of legacy extensions to modern frameworks (Chapter 4). These challenges stem from the differences in programming models and necessity to conform to certain security principles. We present Morpheus, an automated toolchain to port legacy Firefox extensions to Mozilla's Jetpack framework.

- We propose a policy checker framework for Jetpack that allows runtime policy enforcement at a fine-level of granularity (Chapter 4). The modular and extensible architecture of a Jetpack extension together with the policy checker allows the developers to seamlessly add or remove security policies without affecting the rest of the code.

- We demonstrate the effectiveness of these tools by extensive experiments with real-world browser extensions. We report on previously unknown 36 capability leaks that are detected by Beacon in 359 Jetpack extensions and Jetpack code base (Chapter 3). We show the utility of Morpheus by applying it to port 52 legacy extension to modern Jetpack framework (Chapter 4). The effectiveness of policy checker framework at blocking attacks is shown by enforcing 7 security policies in transformed extensions (Chapter 4).

1.7 Contributors to the Dissertation

The following is a list of people who co-authored papers from which material was used in this dissertation. Chapter 3 of this dissertation is the result of a collaboration with my advisor, Professor Vinod Ganapathy, Professor Chung-chieh Shan and my colleague Dr. Mohan Dhawan. During this time, Professor Shan and Professor Vinod Ganapathy contributed to formulating the problem of detecting violations of security principles in Jetpack modules as a capability leak detection problem. Dr. Mohan Dhawan developed modules to automate the evaluation

of the tool and helped in conducting large number of experiments. The idea of automatically porting legacy Firefox extensions to the Jetpack framework, as illustrated in chapter 4, was first coined by Professor Vinod Ganapathy. Dr. Mohan Dhawan developed the initial prototype for rewriting the legacy JavaScript code and designed the module for handling synchronous chrome-content communication against an asynchronous message passing channel.

Chapter 2

Background

The primary purpose of Web browsers is to allow users to locate, retrieve and view content on the Web available in different formats *e.g.,* HTML based Web pages, files, image, videos and audios. Web browsers are also used as clients to update and maintain Web applications that run on top of the browsers. The immense popularity of these applications can be attributed to the ubiquity of the latter across various types of devices. There are more than 80 different browsers available today, with the most popular being Mozilla Firefox [57], Google Chrome [40], Internet Explorer [55], Safari [22] and Opera [68]. The extensible architecture of these browsers allows third-parties to enhance the browser functionality and has been instrumental in making the browser a prominent computing platform. In this chapter, we provide a brief discussion about the essential Web technologies, browser extensions and their security issues.

2.1 Core Web Browser Technologies

The major components of Web browsers include HTML/CSS parsers, layout and rendering engine, JavaScript interpreter, network protocol stack and storage layers. The following is a discussion of some of the standard set of essential Web technologies that forms the core of browser extensions and Web applications.

HTML DOM

All Web documents are primarily written in HTML [32] and CSS [31]. While HTML, a declarative markup language, describes the structure of the Web document, CSS dictates its presentation and style. Elements in the HTML document form a tree structure that is internally represented and can be manipulated by a programming API known as Document Object Model(DOM) [4]. Each HTML tag in a Web page is represented by a node in the DOM tree.

Each DOM node also contains the associated CSS data as well as any application-defined event handlers for GUI activity.

JavaScript

JavaScript [39] is a lightweight, interpreted, dynamic language that has become the *Lingua-franca* for Web and for browser client side scripting in particular. It is prototype-based and object-oriented, and allows functions to be treated as first class objects. It allows Web pages to dynamically modify their HTML DOM structure, CSS style properties and alter the displayed content. It also permits Web pages to register handlers for GUI events that empowers end-users to interact with the page. JavaScript also allows a page to asynchronously fetch data from Web servers and thus provide a rich experience for Web application users.

2.2 Browser Specific Technologies

In addition to the most common technologies discussed above, browser vendors sometimes introduce their own technologies to simplify programming for the concerned platform. What follows is a brief description of two important technologies used in the Mozilla [59] application suite and frequently referenced in the remainder of the dissertation.

XPCOM

Extensions for Mozilla's Firefox browser are mostly implemented in JavaScript which lacks access to system resources. To enable the Firefox extensions to access various system resources, Mozilla uses a technology named XPCOM [66]. XPCOM is a cross platform component object model that has multiple language bindings, allowing XPCOM components to be used and implemented in JavaScript, Java, and Python in addition to C++. XPCOM provides a set of core components and classes, for file and memory management, threads, network access and more, endowing JavaScript based browser extensions to interact with the system-level resources.

XUL

To design portable, cross-platform user interfaces for all applications the Mozilla platform uses XUL (XML User Interface Language) [20]. XUL is Mozilla's XML-based language that includes all the features available in XML and aims to make development for the Mozilla browser easier and faster by simplifying development and modification of the interfaces. XUL allows developers to create common GUIs including various input controls, toolbars with buttons or other content, different types of menus, tabbed dialogs, trees for hierarchical or tabular information and keyboard shortcuts.

2.3 Web Application Security

To enrich users' browsing experience, Web applications often include third-party JavaScript libraries such as jQuery, Google Analytics, Facebook and other social APIs. In such cases, scripts from different domains or origins can potentially interact with each other and, if permitted, can access application data. However, scripts in Web applications are restricted by two security policies — *same-origin policy* [18] and *Content Security Policy (CSP)* [1]. Same-origin policy prevents scripts of different origins from accessing data when they are loaded in different *iframes* [8]. CSP allows Web page authors to whitelist trusted sources of content that browsers can load on a page. Browsers rely on these two security policies for securing Web applications and protecting application data. However, browser extensions are typically oblivious to the origin of content and thus are historically different from Web applications in terms of functionality. Therefore, the security measures for Web applications are not applicable for browser extensions.

2.4 Web Browser Extension

A browser extension adds functionality to the browser itself, often in the form of extra toolbars, context menus, or customizations to the browser's user interface. 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 extensions developed mostly in JavaScript, HTML and CSS. Extensions offer a diverse set of functionalities – displaying

Figure 2.1: A real-world extension displaying weather information.

specific data based on user preference, customizing the rendered Web page, accessing and even manipulating security and privacy sensitive data, developing and debugging Web applications and many more. Figure 2.1 shows an extension that displays weather data upon clicking a particular icon. Some extensions become so popular that they are later integrated with the main browser as an inherent feature. For instance, search-bar or tabbed browsing experience was initially introduced by Mozilla Firefox extensions and later adopted by the next Firefox version. Browser extensions are generally developed by third-party developers, widely available and improve browsing experience for the end-users enabling them to customize the available features by installing various extensions. Extensions help form a developer community for the concerned browser platform and eventually contribute in large part to the popularity of the Firefox browser and related tools, such as the Thunderbird mail client.

As of July 2015, around 14000 extensions, offering a wide variety of functionalities, are available for Firefox via the Mozilla extensions gallery. Popular examples of extensions for Firefox include GreaseMonkey [6], which customizes the look and feel of Web pages using user-defined scripts, Firebug [5], which is a JavaScript code development environment, and NoScript [16], which is a security extension that aims to prevent the execution of unauthorized third-party scripts.

To support diverse functionalities, most modern Web browsers export privileged APIs (*e.g.,* XPCOM for Firefox) allowing extension developers to access sensitive resources like file systems, passwords, cookies, networks and more. This unrestricted access to sensitive resources allows JavaScript(JS) based extensions to run with the privilege of hosting principals *i.e.,* a Web browser. Thus, browser extensions are fundamentally different from Web applications that have limited authority governed by principles like same-origin policy. Therefore, compared to Web

applications these extensions pose more threat to end-users, as the benign-but-buggy extensions can be exploited by remote attackers to take control over the entire Web browser.

This high security risk compelled the browser vendors to maintain authorized sites for hosting extensions. The addon gallery is the official site for Firefox extensions while the Chrome Web Store is the official means for users to find and install extensions for Google Chrome. The addon gallery and Web store is similar to other app stores, such as those for iOS and Android, in that developers make their extension available for users to download. In addition to these authorized places, extensions can also be installed manually by a user or an external program. But unlike the ones on official sites these extensions do not go through a rigorous security review process. In case the developer of such extensions intends to provide a security guarantee, they can use either Beacon or Morpheus. Browsers vendors typically discourage end-users from downloading and installing extensions from untrusted sources due to potential security consequences.

Each browser offers a different extension architecture, extension technologies and API provided for extension programming. For example, in Firefox, extensions can modify the browser UI using a feature of XUL called an overlay, which allows the UI provided from one source, in this case, the Firefox browser, to be merged together with the UI from the extension. In Google Chrome, extensions can use chrome APIs [41] or Web APIs [43] for the same purpose.

2.4.1 Legacy Extension Architecture

Legacy extensions are typically written using open technologies such as HTML, CSS, JavaScript and XUL. These extensions often utilize privileged browser APIs like XPCOM to access system resources and perform useful tasks. In terms of interaction with API, JavaScript code in extensions can be divided into two categories — privileged JavaScript code (*chrome script*) accessing XPCOM and unprivileged JavaScript code (*content script*) interacting with the untrusted Web content on Web pages. However, the original extension architecture used by Firefox has a number of features that make it vulnerable. We briefly discuss some of them below:

- *Unified JavaScript heap*: In Mozilla's legacy extensions, both unprivileged content

scripts and privileged chrome scripts execute in the same heap, raising the risk of shared references. Due to the limitations of the isolation mechanism that attempts to separate untrusted references of the content JavaScript from the chrome JavaScript, the interface has been exploited [25, 71] by attackers. In such cases, if the user navigates to a malicious Web page, the attacker could manipulate the shared object references and influence the execution of the privileged code within the extension. Such scenarios of privilege escalation have previously been used to exploit vulnerable extensions [6].

- *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 can access any system resources and effectively has control over the entire browser. Therefore, the availability of `Components` to all scripts by default significantly increases the chances of vulnerability exploitation in a shared heap environment.

- *Chrome DOM*: The chrome DOM encodes the visual representation of the browser's UI including toolbars, menus, statusbar and icons. Similar to JavaScript code on a Web page that can access the page DOM, chrome scripts also have access to the chrome DOM and can programmatically modify the browser's entire UI.

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, security of legacy extensions predominantly relies on extension developers' discretion and expertise. Much prior work has shown the drawbacks of legacy extensions [24, 25, 35, 36] and highlighted the deficiencies in the design of the legacy architecture.

2.4.2 Modern Extension Frameworks

Web browser vendors have addressed the shortcomings of the legacy extension architecture and recently developed new extension frameworks aimed at better isolating extensions while still allowing them access to privileged browser state. Examples of such frameworks include the Google Chrome extension architecture and Mozilla's Jetpack extension framework for Firefox.

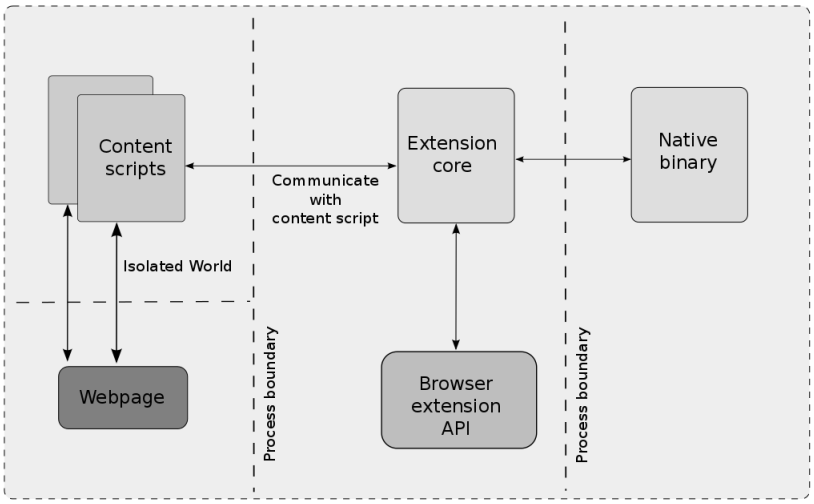

Figure 2.2: Google Chrome extension architecture.

Google Chrome Extension Architecture

Addressing the security issues of legacy Firefox extension architecture, Google Chrome incorporates security principles in designing its own extension architecture [25]. It aims to protect users from vulnerabilities in benign-but-buggy extensions by dividing the extension into three process-isolated components: *content scripts*, *extension core* and *native binary*, and following three security principles : POLA, privilege separation and strong isolation. Figure 2.2 shows a high level overview of the Chrome extension architecture. It features three primary security mechanisms:

- *Privilege separation*: Chrome extensions adhere to a privilege-separated architecture. Each extension typically consists of two types of components: zero or more content scripts and zero or one extension core. Content scripts are limited to only interacting with untrusted Web content and therefore execute with no privileges. Extension core implements extension specific features including browser UI modification, interacting with system level resources via Chrome's extension API and therefore executes with the extension's full privileges. These two types of components, having different level of privileges, are isolated from each other using separate processes. Therefore, content scripts

cannot access any extension API, and similarly, the extension core cannot directly interact with Web sites and need to communicate to content scripts via JSON [13] messages. However, the extension core does not have access to the host machine and an extension can optionally have a native binary component for this purpose. The native binary component can only interact with the extension core via standard Netscape Plugin API (NPAPI) interface and not with content scripts. Such privilege separation between components prevents the attacker-controlled Web sites to gain direct access to the privileged extension API and take control of the browser.

- *Isolated worlds*: Apart from the process-level isolation between components, there is an additional layer of isolation between content scripts and the Web pages it interacts with. This isolated worlds mechanism is intended to protect content scripts from Web attackers by disallowing pointer exchange in a shared program heap between them. For this purpose, even though a content script can access or modify a Web site's DOM, their JavaScript heaps are separate and each have their own DOM objects. This makes it difficult for attackers in malicious Web sites to tamper with content scripts in vulnerable extensions.

- *Permissions*: The goal of permissions is intended to mitigate extension core vulnerabilities. By default, a chrome extension has no access to browser APIs and must explicitly request permissions for such access. Each extension comes packaged with a manifest, an upfront specification of permissions that governs the access to the browser APIs and Web domains. A vulnerable extension, if compromised, cannot request more permission than those already requested explicitly by the developer. These permissions, once granted, allows only the extension core to access the privileged APIs. Consequently, the severity of a vulnerability in an extension is limited to the API calls and domains that the statically declared permissions allow.

Mozilla's Jetpack Framework for Firefox

The *Jetpack framework* [9], officially known as addon SDK [11], focuses on simplifying the extension development process with an emphasis on modular development, code sharing and

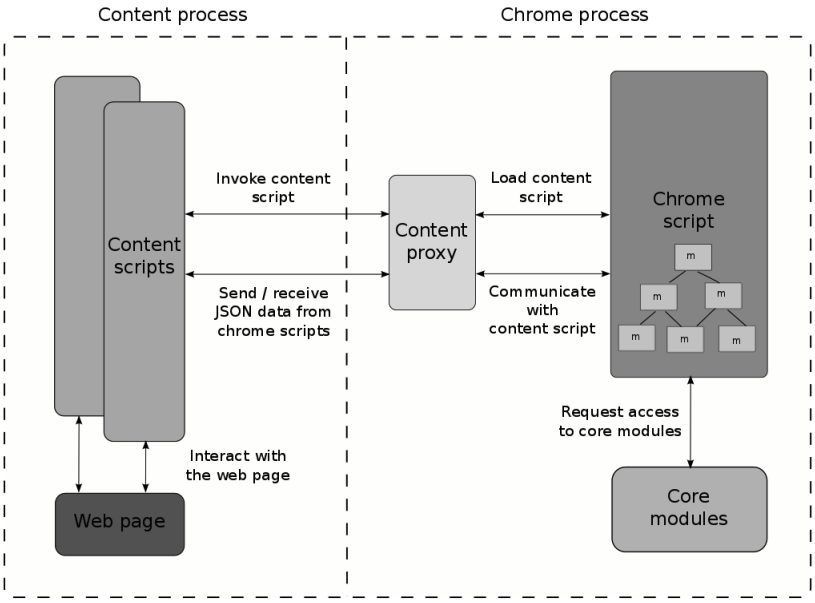

Figure 2.3: Architecture of a simple Jetpack extension.

security. The framework provides high-level APIs, allowing extension authors the ease of writing extensions using standard Web technologies, like JavaScript, HTML and CSS. This is in contrast to traditional extension development, which required developers to be proficient in Mozilla specific technologies like XUL [20] and XPCOM [66].

Figure 2.3 shows the overall architecture of a Jetpack extension. Each extension consists of at least one privileged chrome scripts and zero or more unprivileged content scripts. Content scripts are responsible for interacting with Web page content and are completely isolated from privileged chrome scripts that interact with core modules encapsulating XPCOM interface access and browser DOM manipulation. Chrome and content scripts communicate via asynchronous JSON pipe further making it harder for the Web attacker to launch a successful attack via a compromised content script.

From a security perspective, Jetpack follows the same design philosophy as the Chrome extension architecture. Adhering to the security principles of privilege separation and POLA, Jetpack also implements key security features *e.g.,* a privilege separated architecture, strong isolation and a permission system, Jetpack aims to confine the effect of vulnerable extensions. Jetpack attempts to provide security guarantees at finer granularity. In addition to process level isolation between components, separate heap for content script and Web page content, Jetpack

further structures the extension-specific code as a collection of modules, the combination of all of which is equivalent to an extension core component in a Chrome extension. These modules are isolated and communicate via cleanly defined interfaces. Permissions are granted on a module-basis rather than to the extension as a whole, as opposed to Chrome extensions. Each Jetpack module is thus limited to the permissions or privilege granted to it statically by the manifest provided by the developer. Together with conformance to security principles such module-level permission further attenuate the authority of a compromised extension and thus severity of vulnerability is further reduced compared to Chrome extension. More on this will be discussed on section 4.4 in chapter 4.

Chapter 3

An Analysis of the Mozilla Jetpack Extension Framework

In this chapter, we study the extent to which the Jetpack framework achieves its goals of reducing risks and protecting users from vulnerable extensions. Specifically, we present Beacon, a tool that uses static analysis to study capability leaks in Jetpack modules and extensions. A capability leak happens when a module requests permission to access 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 permission for the interface, thereby breaking modularity and the permission system, and essentially violating the principle of least authority (POLA). Beacon also detects other violations of least privilege or POLA, *i.e.,* cases where a module requests access to an XPCOM interface, but never uses it. A vulnerable module that violates POLA can endow an attacker with more privileges than if the module satisfied POLA. We present the design and implementation of Beacon in detail followed by an evaluation of its effectiveness in real world Jetpack extensions and Jetpack modules itself.

3.1 Problem

As discussed earlier, most modern browsers allow third-party developers to enhance core browser functionality by developing browser extensions. To support rich functionality, Mozilla's Firefox browser exports an XPCOM interface [66] that JavaScript code in extensions can use to access a wide variety of privileged browser objects and services. Access to the XPCOM interface endows JavaScript code in an extension with capabilities that are normally not available to JavaScript code in a Web page. For example, JavaScript code in an extension can freely send `XMLHttpRequest`s to any Web domain, without being constrained by the same-origin policy. The extension 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 extensions [24] found several extensions demonstrating insecure programming practices and exploitable vulnerabilities. A successful exploit against vulnerable extensions 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 extensions is *the lack of development tools for extension authors*. Extension 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 extension. An exploitable vulnerability anywhere in the extension typically exposes the entire XPCOM interface to the attacker.

To address this problem, Mozilla has recently been developing the Jetpack framework [11] that aims to improve the way extensions are developed. It does so using *modularity* and by attempting to enforce *the principle of least authority (POLA)* [70]. A Jetpack extension 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 extension 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 extension 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 extension framework also facilitates code reuse: Jetpack extension authors can contribute the modules used in their extensions to the community, following which others can use the modules within their own extensions. 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 extensions.

In this chapter, 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 extensions. 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 module that violates POLA can endow an attacker with more privileges than if the module satisfied POLA.

For capability flow analysis, we favor static analysis than dynamic analysis even though static analyses are known to be susceptible to both false positives and false negatives. The underlying reason is that we want to avoid well-known drawbacks of dynamic approach — performance overhead and risk of missing any particular program execution path due to typical low code coverage. On top of that, adapting dynamic approaches for this purpose would require instrumentation of both browser and Jetpack extension runtime. In sharp contrast, our static analysis based solution is more lightweight and does not require any modification of either of them. And the extensive evaluation on the real world extension code as discussed later shows that our solution is effective and capture the leaks in a time and memory efficient manner.

3.2 Background and Motivation

The Jetpack framework [9] focuses on easing the extension development process with an emphasis on modular development, code sharing and security. The framework provides high-level APIs, allowing extension 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 [20] and XPCOM [66].

A Jetpack extension is a hierarchical collection of JavaScript modules, with each module exporting some key functionality. A typical Jetpack extension 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 extension 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 extension. 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 POLA [12] when designing modules. This helps in attenuating the capabilities of modules. The modular architecture of a Jetpack extension 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 [23, 25, 35] 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 extension authors could inadvertently attach such capabilities to the `exports` interface. Importing such modules would make the requesting module over-privileged and violate POLA.

Figure 3.1 shows the architecture of a simple Jetpack extension 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 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.

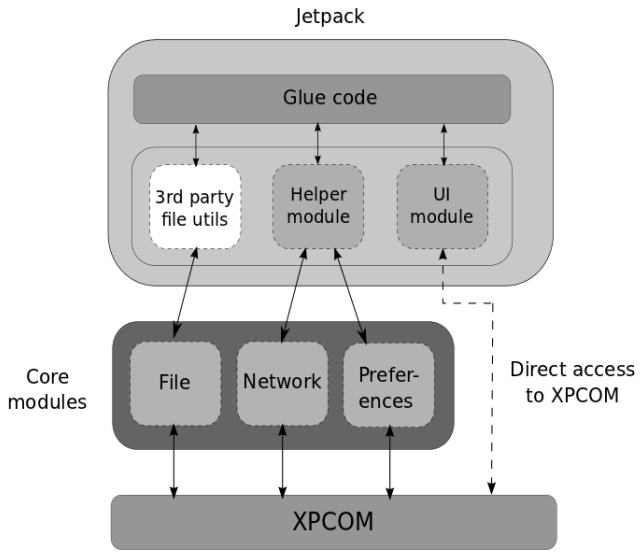

Figure 3.1: Modular Structure of a simple Jetpack extension to download files.

```
(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;
```

Figure 3.2: Code snippet of a module from a real-world Jetpack extension which leaks the capability to access and modify browser preferences.

Capability leak in a Jetpack extension

Consider the code snippet as shown in Figure 3.2 which represents the actual code of the `Preferences` module from 'Customizable Shortcuts' [2], a popular Jetpack extension 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*** 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 extension 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 extension 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

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 3.1: List of some privileged resources and their access interfaces.

choices during module design and implementation. Although adherence to POLA ensures that a module has the minimal set of capabilities required to perform 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.

3.3 Static Analysis of Jetpack Modules and Extensions

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 3.1. 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) [33] 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.

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 3.2, `_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.

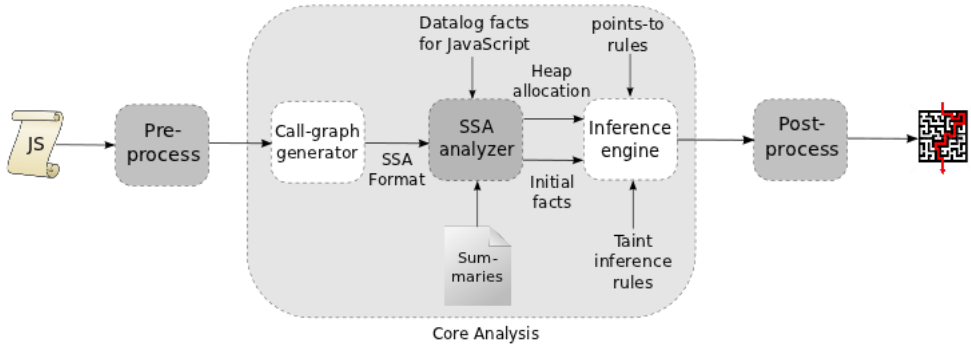

Figure 3.3: Overall workflow of our analysis.

3.3.1 Stages of the Analysis

Our analysis is based upon Datalog and proceeds in three stages. In the first stage, the analysis pre-processes the extension 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 extension code. The results of core analysis are then processed in the third stage to identify offending flows in the source code of the Jetpack extension. Figure 3.3 illustrates a schematic diagram of the analysis of a Jetpack extension. 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:

Pre-processing

Our core analysis (as will be described in Section 3.3.1) 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

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

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 extension 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 3.2 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 [66] 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 3.2 is re-written as shown below:

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

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

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`.

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 [44]. 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 [44, 78], 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 3.4. 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 [23, 44] 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.

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

Table 3.3: Summary of `preferences` module showing the capability leaks.

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 3.3 shows the summary for the code module shown in Figure 3.2.

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 of SSA statements. The analysis engine traverses each of these statements and produces Datalog facts capturing its semantics, as illustrated in Table 3.5. 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 3.4 are generated. The analysis engine then applies the Datalog inference rules presented in Table 3.6 over the initial set of facts to keep track of aliases and the flow of capability through the JavaScript code.

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

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 3.4: Datalog relations used in our static analysis.

which they were externalized. This processed information is also utilized for generating a summary for the analyzed module.

3.3.2 Capability Flow: A Concrete Example

We now demonstrate how the analysis detects capability flows from the `exports` interface of Jetpack extension modules. Figure 3.4 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 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.

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 3.5: Datalog facts generated for each JavaScript statement.

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:

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

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:

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 3.6: Datalog inference rules for points-to analysis.

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

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.

	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, $v_{Preferences}$)

Figure 3.4: Example showing the flow of capabilities through the module's `exports` interface.

On consulting the Datalog inference rules in Table 3.6 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.

3.4 Implementation

We realized the analysis described in Section 3.3 in a tool named Beacon. Beacon is built atop WALA [77], an existing static analysis tool, and uses WALA's capabilities to convert preprocessed 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 [27]. 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.

3.5 Results

We evaluated the effectiveness and accuracy of Beacon in detecting capability leaks by analyzing the entire set of 359 Jetpack extensions 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 3.5.1 we present results from analysis of the capability leaks in core modules and Jetpack extensions. In Section 3.5.2 we study the nature and usage of capabilities in various Jetpack extensions. Lastly, in Section 3.5.3 we report on the use of Beacon to analyze the privileges associated with Jetpack extensions 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 3.3.1 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.

3.5.1 Capability Leaks

Beacon detected 12 capability leaks in four core modules and another 24 leaks in seven Jetpack extensions. 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 3.7 and 3.8 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 3.7: List of capability leaks observed in the core modules. † indicates multiple reference leaks.

Jetpack extensions	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 window	Property of `this` object Return of function attached to `this`	No No
Firefox Share	nsIPrefBranch, window Reference to built-in SQLite database nsIObserverService nsIScriptableInputStream nsIBinaryInputStream nsISocketTransportService nsISocketTransport nsIInputStreamPump Instance of the imported `Socket` module	Property of `this` object Property of `this` object Exported property Return value of exported function Return value of exported function Property of `this` object Property of `this` object Property of `this` object Property of `this` object	No No No No No No No No No
Most Recent Tab	nsIPrefBranch window	Property of `this` object Function return	No No
Open Web Apps	nsIPrefBranch, window Reference to built-in SQLite database nsIObserverService	Property of `this` object Property of `this` object Exported property	No No No
Recall Monkey	nsIIOService nsIFaviconService	Property of `this` object Property of `this` object	No No

Table 3.8: Capability leaks in Jetpack extensions.

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 3.2, 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 extension SDK [11], 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 Extensions

Capability leaks discovered by Beacon in the Jetpack extensions 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 extension uses third-party modules which

themselves leak capabilities. We describe two popular Jetpack extensions which demonstrate all four classes of capability leaks.

- **Customizable Shortcuts**: Customizable Shortcuts is a popular Jetpack extension with over 5000 users. It enables users to easily create keyboard shortcuts to customize the Web browser. We analyzed the extension 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 extension. 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 extension. 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 extension by Mozilla Labs which allows fast and easy sharing of links from any Web page. This extension has 25 modules with over 5300 lines of JavaScript code. Several of these modules have been reused from another Jetpack extension, 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 3.8 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.

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 [35, 37].

- **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.

3.5.2 Capability Use

The Jetpack framework automatically generates a manifest for each Jetpack extension that provides a dossier about the core modules 'required' by the extension, but provides no information about the XPCOM interfaces invoked by the modules in the extension. As revealed in Section 3.5.1, a large number of capability leaks originated from the direct use of XPCOM interfaces. In this section, we analyze the Jetpack extensions and determine the XPCOM-level capabilities associated with them. A concrete understanding of the capabilities associated with a Jetpack extension is useful to both the end-user and Mozilla itself.

- extension reviewers at Mozilla can use capability leak analysis to publish fine-grained Jetpack extension manifests that accurately lists all its capabilities. This would be helpful to end-users in making a well-informed choice when installing an extension. For example, if a Jetpack extension invokes the `nsICookieManager` and also has access to the network, then the end-user can be made aware of the fact that the extension is capable of reading user cookies from all domains and sending them over the network.

- A capability analysis of existing Jetpack extensions 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

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

Table 3.9: Top 10 XPCOM interfaces used in Jetpack extensions.

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

Table 3.10: Top 10 core modules used in Jetpack extensions.

would help the curators at Mozilla to identify extensions 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 extensions conform to the hierarchical model where the developer maximizes the use of the built-in core modules for the Jetpack extension functionality.

To understand the usage pattern of capabilities in Jetpack extensions, we modify Beacon to collect two kinds of capability usage characteristics. First, we track all heap object creations that occur when a Jetpack extension 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 3.5 shows the frequency distribution of XPCOM interfaces for the 359 Jetpack extensions which directly invoke atleast one XPCOM interface. We observe that 46 of the extensions directly invoke XPCOM functionalities, with one Jetpack extension (Firefox share by Mozilla Labs) invoking 14 XPCOM interfaces. Thus over 12% of Jetpack extensions 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.

Tables 3.9 and 3.10 list the top 10 XPCOM interfaces and core modules currently in use by Jetpack extensions. We observe that 5 of the XPCOM interfaces listed

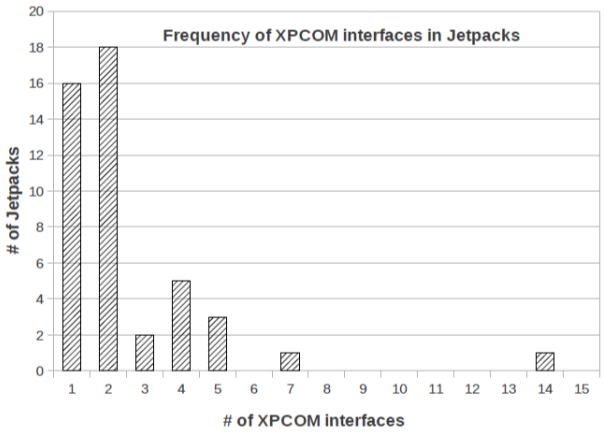

Figure 3.5: Frequency of XPCOM interfaces used in Jetpack extensions.

in Table 3.9, namely `nsIWindowMediator`, `nsIPrefService`, `nsIWindowWatcher`, `nsILocalFile` and `nsIObserverService`, are used by extension 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 extension 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 extensions 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 extensions 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,

Jetpack extension	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 3.11: List of Jetpack modules accessing multiple categories of XPCOM interfaces.

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 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 extensions, where each module invoked XPCOM interfaces to obtain capabilities of different nature. Table 4.5 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.

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 3.12: List of core modules violating POLA.

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 extensions which had direct invocations to XPCOM interfaces. Beacon detected 46 extensions with XPCOM capabilities. The remaining 7 extensions invoked XPCOM interfaces from within event handling code (which Beacon does not model — for reasons stated in 3.3.1).

3.5.3 Over-privileged Modules

The Jetpack extension documentation outlines several guidelines about best practices for developing modules. One of them recommends module authors to follow the principle of least authority (POLA) [12]. 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.

Table 3.12 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 [10].

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 3.3.1, Beacon does not analyze event handling code, thereby causing false positives.

3.6 Summary

In this chapter, 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 [15], Harmony modules [7], SproutCore [19]. However, our focus was on browser extensions implemented using Jetpack. Beacon cannot directly be applied to non-modular extensions.

We implemented Beacon and used it to analyze 77 core modules from Mozilla's Jetpack framework and another 359 Jetpack extensions. 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 extensions. 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 extensions 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 extension 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 extensions 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.

Chapter 4

Porting Legacy Mozilla Extensions to the Jetpack Framework

In this chapter, we present Morpheus, a static analysis and transformation tool that allows legacy Firefox extensions to be systematically ported into Jetpack in a manner that allows enforcement of fine grained security policies without any modification to browser runtime. Porting legacy extensions to Jetpack is necessary since it both improves security of extensions and make them compatible with future multi-process Firefox architecture. However the challenge for the extension developer lies in understanding the key architectural differences between legacy and modern extensions, and in manually transforming the code without altering the basic features.

4.1 Problem

To support extensions, browsers typically expose an API that gives access to privileged browser objects. As discussed earlier, Mozilla's XPCOM API [66] 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. Prior work [24] has uncovered vulnerabilities in the legacy Firefox extension architecture and prevalence of of insecure programming practices in legacy extension 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.

Modern browser extension frameworks address the security issues of legacy extension architecture that can compromise the security of browser itself and therefore aim to better isolate extensions [9, 17, 25, 34] . 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 disallow privileged API access to extension code that interacts with untrusted scripts on a Web page. Consequently, even if an attacker hijacks this portion of the extension, he will be unable to access privileged browser objects. 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 chapter, 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 14000 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 chapter 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.

The rest of this chapter is organized as follows. We start in Section 4.2 by presenting background material on legacy Firefox extensions and new features introduced by the Jetpack framework. We then present the design of Morpheus in Section 4.3, followed by a security analysis in Section 4.4. We then discuss the implementation of Morpheus in Section 4.5, and an experimental evaluation of Morpheus in Section 4.6.

4.2 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.

4.2.1 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 showed in the webpage itself. An attacker can wrest control of 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 *i.e.,* unprivileged content scripts from the JavaScript code executing within the extension *i.e.,* 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.

4.2.2 Legacy Extensions on Firefox

Consider Figure 4.1, 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, the construct `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. Lines 34-36 identify all DOM[1] elements containing user-specified location in the active Web page and invoke `getWeatherData` method defined on the `Weather`

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

```
(1)  function getZipCode(locationStr){
(2)    Components.utils.import('resource://gre/modules/FileUtils.jsm');
(3)    var dir = 'ProfD', filename = 'zip.txt';//'ProfD' is profile directory
(4)    var file = FileUtils.getFile(dir, [filename]);get 'zip.txt' file from 'ProfD'
(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) }
```

Figure 4.1: Code snippet from the DisplayWeather extension.

object to retrieve latest weather updates. The method `modifyWebpageContent` in line 36 actually overlays the weather information on the active Web page.

The above snippet highlights several features commonly 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, code in 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 limitations and a history of exploitable bugs [25, 71]. 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. Such scenarios have previously been used to exploit vulnerable extensions [6].

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 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 [24, 25, 35, 36].

4.2.3 The Jetpack Extension Framework

The Jetpack extension framework [9, 56] 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 [25]. 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 [63].

Conceptually, each Jetpack extension has two parts: (1) at least one 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 [13] 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 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

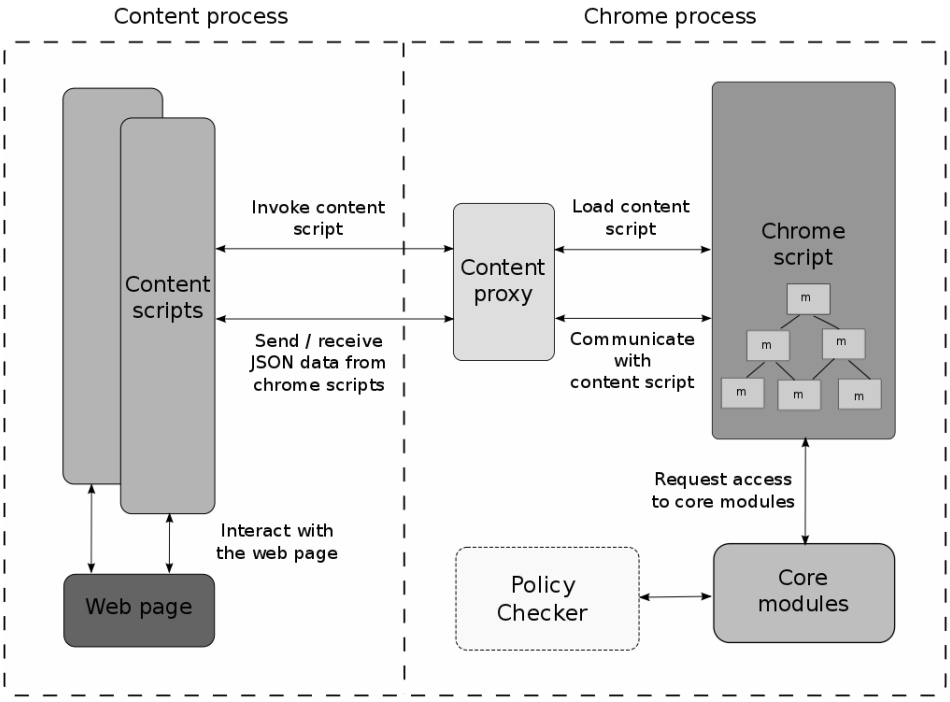

Figure 4.2: Architecture of a simple Jetpack extension. Policy Checker is not part of original architecture and is introduced by Morpheus.

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 explicitly through the module's `exports` interface.

Figure 4.2 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.

4.3 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.

4.3.1 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 4.1) 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 4.3.3 and 4.4,: 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 4.1. 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 [47]. To safeguard core Jetpack modules from becoming confused deputies themselves, (see Section 4.2) 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.

$\mathbb{E}$	Set of all expressions
E_{ρ_f}	Fixed property access expression of the form `e.x`, `e['x']`
E_{ρ_d}	Dynamic property access expression of the form `e[v]`
E_ρ	Property access expression where $E_\rho := E_{\rho_f} \cup E_{\rho_d}$ where $E_\rho \subset \mathbb{E}$
E_μ	Method invocation expression `e.f( args )`, `e['f']( args )`, `e[vf]( args )`
E_χ	XPCOM invoke expression, where $E_\chi \subset \mathbb{E}$. 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_Ω	Object Literal expression of the form `{ a:1, b:function(){}}`, where $E_\Omega \subset \mathbb{E}$
E_d	Function/ variable declaration expression, where $E_d \subset \mathbb{E}$. 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)	$\{e \mid \xi \in (E_\rho \cup E_\mu)\}$ object whose property is accessed in expression ξ
PROPERTY(ξ)	$\{e \mid \xi \in E_\rho\}$ property being accessed in expression ξ
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

Table 4.1: Common notations used in transformation rules and algorithms.

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 leave performance optimization of the rewritten extensions as a topic for future research.

4.3.2 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 [76] over the extension's JavaScript code, and (iii) some basic transformation rules $\mathcal{R}$ (see Table 4.2). Each transformation rule modifies an expression ξ from the program's abstract syntax tree (AST) T. TRANSFORM in

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 nodes for object literals*/`
  $S :=$ COMPUTESENSITIVESET($L, \mathcal{A}$) `/*set of sensitive objects*/`
  $D :=$ COMPUTEDOMSET($L, \mathcal{A}$) `/*set of DOM objects*/`

foreach $n \in$ NODES($\mathcal{T}$) **do**
  $\xi_n :=$ EXPRESSION(n)
  if $\xi_n \in E_\chi$ **then**
    REWRITE($\xi_n, \mathcal{T}, \mathcal{R}1$) `/*rewrite with require, import core modules*/`
  else if $\xi_n \in E_\mu \wedge$ (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 $\xi_n \in E_\rho \wedge$ (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 $\xi_n \in E_\Omega \wedge$ 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.

turn invokes algorithms 4.2(a), 4.2(b) and 3 to complete the transformation. Table 4.1 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 chapter, we refer chrome context simply 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 [76]. The analysis is based on the observation that 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 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 4.2(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')`.

Morpheus addresses a key challenge that arises as a result of the design of Jetpack's

ComputeDOMSet(L, $\mathcal{A}$)
Input: L: Legacy code, $\mathcal{A}$: Alias relation
Output: D : set of 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 $(\xi_n^r \in D)$
$\vee(\xi_n^r \in E_\mu \wedge ((\text{Node}(n, \text{Object}(\xi_n^r)) \in D) \vee \text{InContent}(\text{Object}(\xi_n^r))))$
$\vee(\xi_n^r \in E_\rho \wedge ((\text{Node}(n, \text{Object}(\xi_n^r)) \in D) \vee \text{InContent}(\text{Object}(\xi_n^r))))$
$\vee(\xi_n^r \in E_\rho \wedge ((\text{Node}(n, \text{Object}(\xi_n^r)) \in D) \vee \text{InContent}(\text{Property}(\xi_n^r)))))$ **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 sensitive objects

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

foreach $n \in$ Nodes($\mathcal{T}$) **do**
$\xi_n :=$ Expression(n)
if $\xi_n \in E_\chi$
$\vee(\xi_n \in E_\mu \wedge (\text{Node}(n, \text{Object}(\xi_n)) \in S))$
$\vee(\xi_n \in E_\rho \wedge (\text{Node}(n, \text{Object}(\xi_n)) \in S))$ **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 4.2: Algorithms for computation of set of nodes corresponding to (a) content DOM objects and (b) sensitive objects.

`contentDOM` module. As shown in line 36 in Figure 4.1, 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

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

Table 4.2: Rewrite rules for expressions. Each rule modifies an expression ξ and updates AST T.

`gBrowser.contentDocument` from the active Web page. Morpheus addresses this challenge by creating opaque identifiers (*i.e.,* capabilities) 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 4.5).

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 4.3) that export accessor methods to these privileged objects. These modules export only four properties, namely `id`, `getProperty`,

```
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_;
```

Figure 4.3: Template for secure core module with policy.

`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 4.4 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 4.2(b)) for the property access and methods invoked on corresponding privileged XPCOM 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 4.2 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

```
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_d 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_μ ∧ (OBJECT(ξ_λ) = ξ_{n_i}) then REWRITE(ξ_λ, T^p, ℛ3) ;
        else if ξ_λ ∈ E_ρ ∧ (OBJECT(ξ_λ) = ξ_{n_i}) then REWRITE(ξ_λ, T^p, ℛ2) ;
m := MAKENEWMODULE(𝒯) ;
𝕄 ∪ = m /*construct the main module and add to set 𝕄*/
return 𝕄
```

Algorithm 3: Algorithm for extracting user modules.

tool [51] 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. A simple partition algorithm could place each line of legacy extension code in a separate module. However this approach is

unrealistic since it would lead to excessive amount of intermodule communication which would ultimately hinder the performance of the extension. 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 4.1, 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 4.2, i.e., using either `GlobalGET` or `GlobalSET`.

Preserving Extension UI

As mentioned in Section 4.2.2, 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.

Figures 4.4 and 4.5 show the rewritten statements and extracted user modules on applying Morpheus to our DisplayWeather extension (see Figure 4.1).

4.3.3 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

```
(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));
```

Figure 4.4: Code snippet from *Main* module of the transformed DisplayWeather Jetpack extension. Only statements from Figure 4.1 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*/
```

Figure 4.5: Code snippet from *Weather* module of the transformed DisplayWeather Jetpack extension. Only the statements from Figure 4.1 that are rewritten by Morpheus are shown.

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 4.3. `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.

The `check` API takes as input a tuple (m, p, α), where m is the unique core module name, p is the name of the property accessed or method invoked and α is an optional list of arguments for method invocation, and returns a boolean indicating whether the request was granted or not. Algorithm 4 shows the pseudocode for identifying if the compromised user module in DisplayWeather extension tries to trick the core modules `FileUtils` or `Network` in accessing

a different file or remote server of attacker's choice. The policy also implements a stateful policy check wherein it detects an access corresponding to a low confidential sink taking place after access to a high confidential source. An example of such a policy is to disallows all network access after accessing the *LoginManager* interface, which is an XPCOM interface that allows to read all stored passwords from the Firefox profile.

```
CHECK(m, p, α)
Input: m : Module name, p : Property, α : ArgumentList
Output: b, where b ∈ {true, false}
if m ="FileUtils" ∧p ="getFile" then
    f := GETARGVAL(m, p, "filePath")
    if f ≠ f^v then return true /*f^v is the permitted filePath*/;
else if m ="XMLHttpRequest" ∧p ="open" then
    u := GETARGVAL(m, p, "url")
    if u ≠ u^v then return true /*u^v is the permitted server url*/;
/*For a low-confidentiality(high integrity) sink, find corresponding
high-confidentiality(low-integrity) sources that would violate any of the
encoded policy*/
if ISSINKACCESS(m, p, α) then
    Γ := FINDSETOFVIOLATINGSOURCE(m, p)
    foreach (m_i, p_i, α_i) ∈ Γ do
        if (m_i, p_i, α_i) ∈ Λ then return true ;
Λ := Λ ∪ (m, p, α) return false
.
```

Algorithm 4: Checking if an access in core module violates policy. Λ is the list of access i.e. Λ :={(m, p, α)} and GETARGVAL(m, p, *str*) gives the value of the argument *str* in $m.p$ invocation.

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.

4.4 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 4.6(b) and 4.6(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 4.6(b). However when processed by Morpheus, `s.p` is rewritten as `s.getProperty('p')` and the heap object

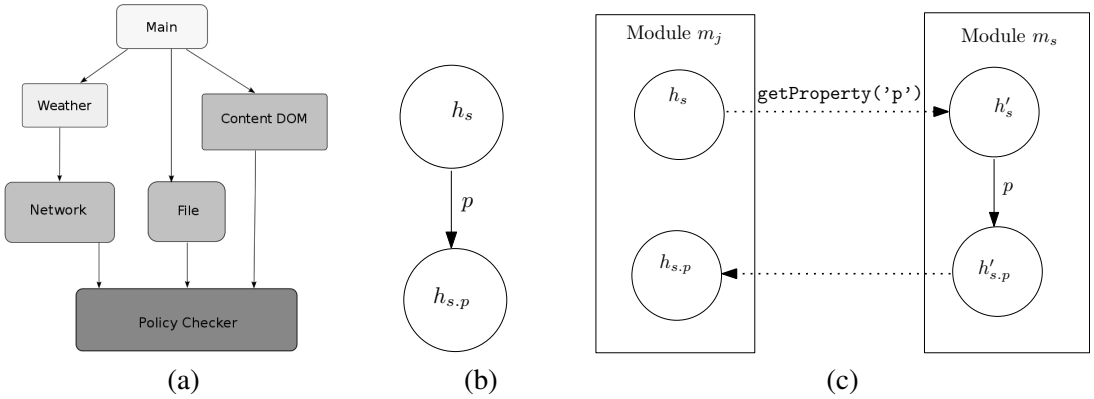

Figure 4.6: (a) Module hierarchy in transformed DisplayWeather extension. Difference of heap map of property access of a sensitive object where h_ξ is the heap object for the expression ξ. (b) `s.p` in legacy extension (c) `s.getProperty('p')` in Jetpack. m_j is a user module, m_s is a module wrapping sensitive object `s`.

.

h_s for `s` does not have direct access to $h_{s.p}$, as shown in Figure 4.6(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 4.3), 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

#	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 cannot 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.

Table 4.3: Security properties.

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 4.6(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 exact 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 4.6(a) shows the module hierarchy for the transformed Jetpack extension. Using the above formula and the transformed code (Figures 4.4 and 4.5), 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.

4.5 Implementation

We realized the entire Morpheus toolchain in about 13,400 lines of JavaScript (node.js [15]), 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 [3], 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 [60] 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 extension 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 4.6.

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 4.2.3). 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 4.1). 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.

4.6 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 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.

4.6.1 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 4.4 lists the extensions evaluated along with their functionality. The top 50 entries are for the real-worls 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.

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

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

Table 4.4: Legacy extensions transformed using Morpheus and corresponding Jetpack statistics.

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.

4.6.2 Conformance to POLA

We used an off-the-shelf tool Beacon [51] to check whether modules in a Jetpack extension adhere to the principle of least authority. Beacon determines whether a Jetpack module leaks

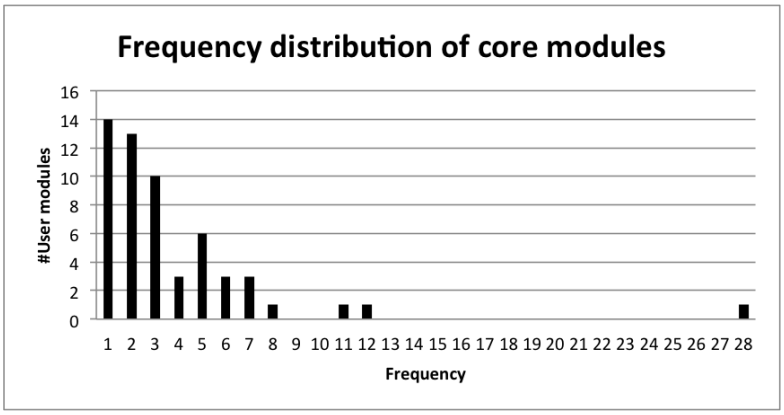

Figure 4.7: Frequency of core modules in Jetpack user modules.

references to privileged objects that it encapsulates. If so, any other code that `requires` 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 resulting in absence of POLA violation.

4.6.3 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 4.7 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 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 4.3.2).

Similar to the categorization of XPCOM interfaces in chapter 3, we further grouped core modules into 6 categories — namely Application, Browser, DOM, Security, I/O, and Miscellaneous, to assess the nature of functionality offered by the user modules. Modules that access application or user preferences, create application threads, etc. are categorized under Application. The category Browser 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 DOM. Modules that handle browser permissions and cookies are grouped under Security, while those that access network, file system or storage come under I/O. The remaining modules are grouped under Miscellaneous. Table 4.5 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 4.3.2) responsible for access to the content objects. Instead they access properties of either chrome `window` or `gBrowser`, which in turn invoke the `contentDOM` to make a transition to the `content`. Because of this implicit invocation, column entires in category DOM are empty for such Jetpack extensions.

4.6.4 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 4.6 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

Jetpackextension	Module name	Categories					
		Application	Browser	DOM	I/O	Security	Misc.
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	M-1	✓	✓	✓			
Button	M-2	✓	✓	✓			
Facebook Touch	M-1	✓	✓	✓			
Panel	M-2	✓	✓	✓			
FlagFox	flagfox (JSM)	✓	✓	✓	✓	✓	✓
	ipdb (JSM)				✓		
	main	✓					
FlashBlock	M-1	✓	✓		✓		✓
	M-2	✓					
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				✓	✓	

Table 4.5: List of Jetpack modules accessing multiple categories of core modules. 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.

our dataset, only Steal-login exhibits malicious activity, while the others are benign and do not violate the policies in Table 4.6. 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

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

Table 4.6: List of policies checked for evaluation data set.

extensions to elicit 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. Even though we introduce a runtime policy enforcement, it does not introduce any perceivable runtime overhead. This is because policy checker is only consulted for the part of code that invokes 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**) *Event handling with chrome/content separation*: An extension from our dataset CommentBlocker[2] 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).Morpheus transforms the code to install a new event handler on the element in the `content`. This newly

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

Original extension code which installs chrome handler in content code.

```
document.addEventListener('DOMNodeInserted', function(evt) {
  CommentBlocker.parser.parse(evt.originalTarget,true);
},false);
```

```
function(e) {
```
This handler is installed instead of CommentBlocker.parser.parse.
```
  var evt = {
    type:  e.type,
    originalTarget:  getOpaqueID(e.originalTarget),
    target:  getOpaqueID(e.target),
    /* add more fields to the evt object */
  };
  self.port.emit('response', JSON.stringify(evt));
}

worker.port.on("response", function (x) {
```
Supporting chrome code.
```
  var evt = createEventObject(JSON.parse(x));
  var evtH = getSavedReferenceToOriginalHandler();
  // evtH is a reference to CommentBlocker.parser.parse.
  evtH(evt);
});
```

Figure 4.8: Transformation required to support execution of `chrome` handlers from event listeners installed in `content` code. Objects `worker` and `self` are used to communicate between chrome and content processes.

installed event handler then performs a context switch into the `chrome` code to execute the event handler in the `chrome` context. This transformation also requires creating opaque identifiers for the "event" object (which is passed implicitly to all handler functions) and transmitting it to the `chrome` for correct execution. 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 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`).

Figure 4.8 shows the snippet of code required for the transformation for the CommentBlocker extension. We observed a perceptible performance slowdown for CommentBlocker when hiding comments. While it is well documented that mutation DOM event listeners themselves considerably degrade the performance of subsequent DOM operations [64], we believe that repeated context switches between `chrome` and `content` also contributed significantly to the overhead.

(**2**) *QueryInterface*: The interface definitions for most XPCOM APIs inherit from other interfaces. For example, the `nsILocalFile` interface inherits from `nsIFile`. `QueryInterface` [61] 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**) *String objects*: 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 to facilitate all string operations.

4.7 Limitations

In its current form, Morpheus is constrained by a number of factors. We enumerate them below:

(**1**) *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 [76] 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. The above issue can be resolved by porting Morpheus to use a more stable platform, like SpiderMonkey [62], to remove such limitations and allow evaluation of more complex extensions.

(**2**) *Jetpack framework*: The Jetpack framework 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`).

(**3**) *Plethora of XPCOM interfaces*: The legacy extension architecture provides myriad XPCOM interfaces for the Mozilla suite of applications. There exist a total of over 1500 XPCOM interfaces[3] available to developers in Firefox. In absence of an automated generation of core modules, each XPCOM interface is processed manually and corresponding core module is developed by hand.

(**4**) *Binaries in custom XPCOM interface*: Firefox also allows developers to define an XPCOM interface by providing an interface definition file and compiling it into a binary format (XPT) in order to be registered and used within the browser. In its present form, Morpheus does not process extensions that includes XPT files.

4.8 Summary

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.

[3] We counted the number of available XPCOM interfaces by iterating over the `Components.interfaces` object.

Chapter 5

Related Work

There has been much interest recently in the research community to improve defenses against vulnerable and malicious browser extensions. To our knowledge, the work presented in this thesis is the first one to do a rigorous analysis of the Jetpack framework, and aims at reducing developers' and reviewers' effort in securing both the modern and legacy extensions. Towards the goal of facilitating secure extension development, it presents the first capability leak detection tool, the first policy checker framework for modern extensions and the first automated approach to port legacy extensions to secure, modern platforms.

5.1 Securing Browser Extensions

Sabre [35] and Djeric and Goel [37] 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 extensions 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 [23, 24] 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 extensions 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 extensions. While VEX was originally applied to traditional Firefox extensions, 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

a lower-precision analysis, Beacon is able to find real vulnerabilities in Jetpack modules and extensions.

LvDetector [80] takes a hybrid approach to automatically detect information leakage vulnerabilities in JavaScript-based extensions. It first uses scenario-driven execution traces to dynamically construct a call graph and then statically analyze the call graph to capture its corresponding information flows. The dynamic call graph construction scheme reduces the overall false positives in the analysis results. In contrast, both Beacon and Morpheus take a pure static approach. Unlike LvDetector, Beacon detects capability leaks in Jetpack modules.

Kashyap and Hardekopf [52] introduces a new notion of addon security signature in order to automate the extension vetting process. A security signature of an extension essentially captures the API usage and detailed information flow and assists in the review process by enabling the reviewer to compare the signature with the extensionś advertised functionality. The proposed approach is still reliant on the reviewer's expertise to decide about the potential security violation and detect the program point where the actual violation occurred. In contrast, Beacon automatically identifies program points that lead to vulnerability. However, theses security signatures can help check policy violations at compile time and reduce the load from the runtime policy enforcement module.

IBEX [46] 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 the module interface. IBEX also requires extensions to first be written in a dependently-typed language (to make them amenable to verification), afterwards they are translated to JavaScript. In contrast, Beacon works directly with Jetpack extensions written in JavaScript.

Runtime policy enforcement has also been applied to prevent extensions from leaking sensitive data and limiting extension privilege [67, 74]. Even though the approach presented in SENTINEL [67] is more light-weight than the one proposed by Ter Louw *et al.* [74], both techniques require modifications to the browser. Similar to Morpheus, SENTINEL wraps all accesses to XPCOM interfaces in legacy extensions to validate the operations with regard to

security policies specified on the extension. In contrast, Morpheus's 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 fine-grained security policy enforcement without modifying browser or Jetpack runtime. To our knowledge, Morpheus is the first tool that retargets legacy extensions to modern frameworks.

The Jetpack framework is similar to the Google Chrome extension architecture [25] which encourages a modular design, as discussed in section 2.4.2. Recent works [28, 54] explore the design of the Google Chrome extension architecture to highlight its deficiencies in developing secure Google Chrome extensions. More large scale studies [50, 75] reveal the prevalence of malicious Google Chrome extensions and highlighted the limitations of the architecture in protecting user privacy. This inspires the preliminary design of a new extension architecture [48]. The proposed design relies on mandatory access control based confinement, similar to COWL [72] — a JavaScript confinement system for Web applications, to prevent sensitive information from being leaked by malicious extensions. The key idea of restricting the actions of an extension based on resource access is similar to our proposed policy enforcement in Morpheus. However, Morpheus relies on developers to write the security policies and the policy checker is independent of the extension code. On the other hand, the new architecture proposes to have the security policies ingrained in the extension APIs.

5.2 Static Analysis of JavaScript

More generally, there has been much recent work on static analysis of JavaScript code executing on Web pages. The dynamic nature of JavaScript makes it hard to analyze it statically. Beacon borrows and builds upon the techniques introduced in these papers (discussed below), but applies them to the analysis of the Jetpack framework. On the other hand, Morpheus utilizes an existing static analyzer for precise information flow analysis of legacy extensions.

The core analysis of Beacon is most similar to that of Gatekeeper [44]. While Gatekeeper was originally applied to study the security of small JavaScript-based widgets, we applied Beacon to study capability leaks in Jetpack. Actarus [45] 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, which has well defined sources (`require` and XPCOM) and sinks (`exports`) for each module. ENCAP [73] 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 [30], 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`.

CFA2 [76] is a new flow analysis for functional languages that provides precise dataflow information. Morpheus uses the CFA2 implementation for JavaScript built into Doctor JS [3], Mozilla's suite of static-analysis tools for JavaScript. Morpheus uses the CFA2 implementation in Doctor JS to analyze legacy extension code.

Feldthaus *et al.* [38] propose a framework for user-driven general-purpose JavaScript refactoring (renaming, property encapsulation, modularization) using static pointer analysis. In contrast, Morpheus only performs a specific refactoring to extract user modules, without any user intervention, towards automatically porting legacy Firefox extensions to the more secure Jetpack framework.

Static analysis of JavaScript code is also conducted in domains other than web application and browser extension. To overcome the problem of supporting cross-platform mobile apps, a HTML5 based technique, which uses standard web technologies like HTML5, CSS and JavaScript, is gaining popularity among mobile app developers. Jin *et al.* [49] characterizes the code injection attacks in HTML5 based mobile apps and performs static analysis of JavaScript in order to detect such attacks.

5.3 Privilege Separation

Morpheus is most closely related to Privtrans [26] and Swift [29]. 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 [53, 65, 69, 79, 81], because it is entirely automatic and does not require any user annotations to accomplish partitioning.

Akhawe *et al.* [21] propose a new architecture to achieve privilege separation for HTML5 web applications including browser extensions. While their primary goal is to design such an architecture, Morpheus converts legacy code to make it compatible with the Jetpack framework that mandates chrome-content privilege separation.

Chapter 6

Conclusion

The flexibility of running third-party browser extensions contributes to the popularity of Web browsers, unfortunately, it leaves the browser open to security breaches caused by vulnerable extensions. To safeguard the browser from extension vulnerability and protect end-user data, Web browser vendors recommend extension developers to adhere to safe programming guidelines. Adherence to these guidelines helps vulnerable extensions limit the damage even if they are compromised. However, browser vendors transfer this burden to third-party extension developers whose discretion and expertise are required to adhere to these guidelines. Furthermore, unsafe programming practices are hard to detect via manual inspection. Therefore, in order to avoid the propagation of detrimental effect of vulnerability, extension developers need diligent skillfulness and security software tools necessary to identify unsafe programming practices; today, even if they have the former, they lack the latter. This dissertation aims to simplify the development of secure browser extensions by aiding extension developers with the tools necessary to comply with recommended safe programming guidelines. It begins with a study on Jetpack, a modern extension framework for the Firefox browser, and presents Beacon for automated detection of unsafe programming practices within extension code. It then presents Morpheus to transform vulnerable legacy extensions to Jetpack-compatible code that improves extension security. Morpheus augments Jetpack with a modular policy checker framework that enables fine grained security policy enforcement. It also allows extension developers to write legacy-style imperative code and automatically port it to Jetpack to benefit from the enhanced security guarantee offered by the framework. Program analysis and software engineering techniques have been used for implementing these tools. Experimental evaluation of both of these tools indicate that they are effective at identifying unsafe programming practices and transform vulnerable legacy code to secure modern code.

6.1 Reflections and Short-term Directions

Both Beacon and Morpheus rely on static program analysis and hence inherently suffer from false positives stemming from JavaScript dynamic constructs. Performance of these two tools can be improved by augmenting them with dynamic taint tracking as a second phase of analysis. To reduce the performance overhead introduced by the dynamic analysis, only code generated by the dynamic constructs can be analyzed.

Instead of leveraging the Datalog query engine, Beacon could have used WALA or a more lightweight DoctorJS for the alias analysis. However, since Beacon keeps track of taint values in addition to aliases, it would require modifications of both the tools as they did not provide any direct interface for such purposes. In particular, WALA, which is a huge JAVA library, changes in the existing analysis related codebase would become cumbersome. On the other hand, in Datalog, realizing the inference rules for capability flow analysis and performing queries were more straightforward and easy to implement.

Performance optimization of the transformed extension is a key aspect that this dissertation does not address for Morpheus. This concern can be addressed by reducing the number of context switches between the chrome and the content. For example, with intelligent rewriting we can perform the evaluation of `gBrowser.contentDocument.location.href` in one context switch instead of three required by Morpheus.

The generation of user modules in Morpheus can be improved by clustering functions based on access to XPCOM functionality instead of the ”owning” object. The current algorithm tries to encode the developer's way of partitioning code and may thus provide less meaningful partitions if the developer does not arrange his code using purposeful object hierarchies.

Morpheus can be extended to make use of core modules provided with the Jetpack framework. To do this, we would need to establish functional equivalence between the legacy interfaces and the modules provided by the Jetpack framework. Code comprehension for abstracting program behavior and statistical API mapping techniques can be explored for establishing functional equivalence between different JavaScript code fragments.

Lastly, Morpheus still relies on extension developers' assistance for encoding extension-specific security policies. A more effective approach could be designing a high-level language

for writing such policies. These policies could then be compiled down to JavaScript code that encode them in the transformed Jetpack extension. An ideal solution would completely remove the extension developer from this workflow and automatically infer the policies by analyzing the extension code itself. An alternate approach could be to tackle the problem at the framework level and redesign the extension API such that the security policies are ingrained in the API itself.

6.2 Other Applications

The techniques presented in this thesis are generic and have other applications as well. The same approach can be extended to investigate difficulties involved in programming for various other extensible platforms such as smartphones, smart cars, smart homes, smart watches and other wearable devices — especially given the fact that these platforms also support applications, popularly known as apps, implemented in HTML5 and JavaScript.

6.2.1 Enhancing Security and Simplify Programming for Extensible Platforms

Every platform comes with a unique challenge of its own and differs in threat model and attack vector. However, some of the existing frameworks for these extensible platforms have a similar permission-based security model as the Jetpack extensions. Hence, techniques presented in chapter 3 can be useful in performing a rigorous security review of the framework.

Exploring programming models for these new platforms and characterizing the necessary modifications , both in app code and UI, in order to retarget existing web and smartphone apps to these platforms, will be an interesting challenge. With increasing support for E6 Harmony modules [7] in most major Web browsers, the current practice is to either write JavaScript directly using modules or automatically transform non modular JavaScript code to make use of modules. It would be interesting to study the benefits and challenges involved in automatic refactoring of legacy monolithic web application code into JavaScript code that uses Harmony modules. Techniques presented in chapter 4 can be useful in this regard.

6.2.2 Browser Based OS

Browser has emerged as a new full-fledged computing platform replacing the traditional operating system, where applications running on top of it are simply web applications written in HTML5, CSS and JavaScript. Two such currently available systems are Firefox OS [58] and Chrome OS[42]. A developer community has already formed to write apps for Firefox OS which directly boots to a rendering engine and is designed to be used in smartphones. This opens a new era of computing the security and programming aspects of which still needs to go through rigorous study. Given that Firefox OS follows the same permission-based security model as the Jetpack framework, similar study, as presented in chapter 3, can be conducted for Firefox OS.

Bibliography

[1] Content security policy. https://developer.mozilla.org/en-US/docs/Web/Security/CSP/Introducing_Content_Security_Policy.

[2] Customizable shortcuts. https://addons.mozilla.org/en-US/firefox/addon/customizable-shortcuts/L.

[3] Doctor JS. http://doctorjs.org/.

[4] Document object model. www.w3.org/DOM.

[5] Firebug: Web development evolved. http://getfirebug.com.

[6] Greasespot: The weblog about Greasemonkey. http://www.greasespot.net.

[7] Harmony modules. http://wiki.ecmascript.org/doku.php?id=harmony:modules.

[8] Html iframe. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.

[9] Jetpack. https://wiki.mozilla.org/Jetpack.

[10] Jetpack addon refactoring oversights. https://github.com/mozilla/addon-sdk/pull/291.

[11] Jetpack sdk. https://addons.mozilla.org/en-US/developers/docs/sdk/1.3/.

[12] Jetpack security model. http://people.mozilla.com/~bwarner/jetpack/components.

[13] JSON. http://www.json.org/.

[14] Mozilla addon gallery. https://addons.mozilla.org.

[15] node.js. https://nodejs.org.

[16] NoScript—JavaScript blocker for a safer Firefox experience. http://noscript.net.

[17] Safari extensions. https://developer.apple.com/library/safari/documentation/Tools/Conceptual/SafariExtensionGuide/Introduction/Introduction.html.

[18] Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[19] Sproutcore. http://sproutcore.com/.

[20] Xul. https://developer.mozilla.org/En/XUL.

[21] Devdatta Akhawe, Prateek Saxena, and Dawn Song. Privilege separation in HTML5 applications. In *Proceedings of USENIX Security '12*.

[22] Apple. Safari. https://www.apple.com/safari/.

[23] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *Usenix Security*, 2010.

[24] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vetting browser extensions for security vulnerabilities with VEX. *CACM*, 54(9), September 2011.

[25] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of 13th Network and Distributed System Security Symposium, NDSS '10*. The Internet Society, 2010.

[26] David Brumley and Dawn Song. Privtrans: automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, SSYM'04, 2004.

[27] Rafael Caballero-Roldn, Yolanda Garca-Ruiz, and Fernando Senz-Prez. Datalog educational system. `www.fdi.ucm.es/profesor/fernan/des/`.

[28] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the USENIX Security '12*. USENIX Association.

[29] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. *SIGOPS Oper. Syst. Rev.*, 41(6).

[30] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow in JavaScript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[31] World Wide Web Consortirum. CSS. `http://www.w3.org/Style/CSS/Overview.en.html`.

[32] World Wide Web Consortirum. HTML. `http://www.w3.org/html/`.

[33] 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.

[34] dev.opera.com. Opera extensions. `http://dev.opera.com/extension-docs/`.

[35] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript based browser extensions. In *ACSAC*, 2009.

[36] Vladan Djeric and Ashvin Goel. Securing script-based extensibility in web browsers. In *Proceedings of USENIX Security'10*.

[37] Vladan Djeric and Ashvin Goel. Securing script-based extensibility inweb browsers. In *Usenix Security*, 2010.

[38] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2011.

[39] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media Inc., April 2011.

[40] Google. Chrome. `https://www.google.com/intl/en/chrome/browser/`.

[41] Google. Chrome APIs. `http://developer.chrome.com/extensions/api_index.html`.

[42] Google. Chrome OS. `http://www.chromium.org/chromium-os`.

[43] Google. Web APIs. `http://developer.chrome.com/extensions/api_other.html`.

[44] Salvatore Guarnieri and Benjamin Livshits. GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security*, 2009.

[45] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011.

[46] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy*, May 2011.

[47] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[48] Stefan Heule, Devon Rifkin, Alejandro Russo, and Deian Stefan. The most dangerous code in the browser. In *Proceedings of the 2015 Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[49] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[50] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Proceedings of the USENIX Security*, 2014.

[51] Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-Chieh Shan. An analysis of the Mozilla Jetpack extension framework. In *Proceedings European Conference on Object-Oriented Programming*, June 2012.

[52] Vineeth Kashyap and Ben Hardekopf. Security signature inference for javascript-based browser addons. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2014.

[53] Douglas Kilpatrick. Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284. USENIX, 2003.

[54] Guanhua Yan Lei Liu, Xinwen Zhang and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *Proceedings of the 19th Network and Distributed System Security Symposium*, 2012.

[55] Microsoft. Internet Explorer. `http://windows.microsoft.com/en-us/internet-explorer/browser-ie`.

[56] Mozilla. Add-on SDK. `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/`.

[57] Mozilla. Firefox. `http://www.mozilla.org/en-US/firefox/`.

[58] Mozilla. Firefox OS. `http://www.mozilla.org/en-US/firefox/os/`.

[59] Mozilla. Mozilla. `http://www.mozilla.org/en-US//`.

[60] Mozilla. Narcissus. http://mxr.mozilla.org/mozilla/source/js/narcissus/.

[61] Mozilla. Query Interface. https://developer.mozilla.org/en-US/docs/XPCOM_Interface_Reference/nsISupports#QueryInterface%28%29.

[62] Mozilla. Spidermonkey. https://developer.mozilla.org/en/SpiderMonkey.

[63] Mozilla Developer Network. Electrolysis. https://wiki.mozilla.org/Electrolysis.

[64] mozilla.dev.platform. Performance impact of dom mutation listeners. http://groups.google.com/group/mozilla.dev.platform/browse_thread/thread/2f42f1d75bb906fb.

[65] Andrew C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[66] Mozilla Developer Network. Xpcom. developer.mozilla.org/en/XPCOM.

[67] Kaan Onarlioglu, Mustafa Battal, William Robertson, and Engin Kirda. Securing legacy firefox extensions with sentinel. In *Proceedings of the 10th International Conference on Detection of Intrusions and Malware*, 2013.

[68] Opera. Opera. http://www.opera.com/.

[69] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, SSYM'03, 2003.

[70] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[71] Addon SDK. Content proxy. https://addons.mozilla.org/en-US/developers/docs/sdk/latest/dev-guide/guides/content-scripts/accessing-the-dom.html.

[72] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2015.

[73] Ankur Taly, Ulfar Erlingsson, Mark S. Miller, John C. Mitchell, and Jasvir Nagra. Automated analysis of security-critical javascript apis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[74] Mike Ter Louw, Jin Soon Lim, and V.N Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4, 2008.

[75] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[76] Dimitrios Vardoulakis and Olin Shivers. CFA2: a context-free approach to control-flow analysis. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, 2010.

[77] IBM Watson. Watson libraries for analysis. `wala.sourceforge.net/wiki/index.php/Main_Page`.

[78] 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.

[79] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3), August 2002.

[80] Rui Zhao, Chuan Yue, and Qing Yi. Automatic detection of information leakage vulnerabilities in browser extensions. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*, 2015.

[81] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using Replication and Partitioning to Build Secure Distributed Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.